Report No. 91022

Report No. 91022

ROYAL SIGNALS AND RADAR ESTABLISHMENT,
MALVERN

AN EXAMPLE MACHINE USED FOR DEVELOPING A PROOF STRATEGY FOR SECURE SYSTEMS

Author: P F Terry†

† Capella Research Ltd
29 Sandown Close, Blackwater
Camberley, Surrey GU17 0EN

PROCUREMENT EXECUTIVE, MINISTRY OF DEFENCE
RSRE
Malvern, Worcestershire.

April 1991

CONDITIONS OF RELEASE

0107298 304043

************************** DRIC U

************************** DRIC Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Report 91022

Title: An Example Machine used for Developing a Proof Strategy for Secure Systems

Author: P F Terry†

Date: April 1991

Abstract

This report describes a machine which is an abstraction of the archetypal Command, Control, Communications and Information (C^3I) system which system developers meet in procurement requests, operational requirements, invitations to tender, etc, from government and military agencies.

The purpose of this report is to set the scope of complexity of structure, functionality and policy which we believe the SMITE approach to secure systems development can encompass. It thus provides background and motivation for future research and encourages those involved in secure systems procurement to investigate further the SMITE approach.

The Abstract Machine is first described in English with pictures and subsequently in the Z specification language.

This report has been furnished for use by Her Majesty's Government under the terms and conditions of contract SLS42c/720 from the Defence Research Agency, Electronics Division.

† Capella Research Limited
29 Sandown Close
Blackwater
Camberley
Surrey, GU17 0EN

Contents

1. Introduction

This document describes a machine which is an abstraction of the archetypal Command, Control, Communications and Information (C^3I) system which system developers meet in procurement requests, operational requirements, invitations to tender, etc, from government and military agencies.

The machine is abstract in the sense that it generalises the features found in such machines and not in the sense that it simplifies the features to the point of non-existence.

For example, real machines have a complex structure, a persistent storage structure, usually a hierarchy for filing systems but potentially attaining the complexity of a relational database, which interacts with a transient process structure, user processes and sub-processes, system services, etc. In abstract models, such as formal policy models, this may be reduced ad absurdum to a subject/object matrix. The abstract machine of this document retains the essential structure evident in real systems.

Similarly, in real machines there are many different types of accesses to many different types of objects, not all of which can be abstracted to simple notions of read access and write access. The abstract machine maintains this fundamental diversity of access while abstracting out unnecessary repetition. Hence, reading a document, reading a draft document, reading a telex, etc, may all be represented by a single read file operation but the fundamental distinction between reading a file and "reading" a directory might be maintained.

Finally, real procurement machines tend to have complex policy requirements, no less rigourously required to be correct, in addition to the simple no-flows-down concept of abstract lattice models of security. The abstract machine we describe, in addition to the usual no read up, no write down checks, embodies a simple abstract implementation of separation of duties which we believe allows such issues as downgrade to be addressed with equal assurance.

The document describes the Abstract Machine first in English with pictures and subsequently in the Z specification language. The latter simply serves to remove ambiguity inherent in any English description of a machine, it does not in any sense serve to prove the security of the machine described. For such a formal statement of the specification and policy together with a proof that the machine upholds the policy see [CRL-720-TR-06/A].

The purpose of this document is to set the scope of complexity of structure, functionality and policy which we believe the SMITE approach to secure systems development can encompass. It thus provides background and motivation to the subsequent reports and encourages those involved in secure systems procurement to investigate further the SMITE approach.

While we positively encourage developers to investigate our approach as a better route to high-assurance system development a note of caution is in order lest we raise expectations to unrealistic levels.

The abstract machine described in this document is undoubtedly more realistic than other specifications which make the same claims for proof assurance. However, it is still a design not an implementation. There are thus still many hurdles to cover. We do not claim to have removed all of these, we just claim that we are more advanced than extant approaches.

On the other hand, the machine described herein is not the absolute limit of complexity which we believe the SMITE approach can handle. If a machine for which one wished to apply this approach exceeded the capabilities of the machine described herein, one should not take this as evidence that the SMITE approach is not applicable. The machine described herein was chosen on the basis of exhibiting a coherent, balanced set of capabilities which capture the bulk of the features seen in most procurements and which constitutes a sufficient advance over extant approaches to justify the expense of switching to a "new" approach. It was also chosen to remain sufficiently simple that we can demonstrate the tractability of the approach, by actually doing the proofs, within the bounds of a research project. We are fairly confident that relaxations of the constraints in the abstract machine can be accommodated within the approach but it is our intention to only claim and describe such extensions as we demonstrate their tractability. The abstract machine therefore should be regarded as our first "stick in the ground" which forms the basis for further incremental research and investigation.

In the attempt to motivate the machines structure and functionality in the English overview the various components are described with examples drawn from the usual melee of entities which developers are used to meeting in procurement descriptions of such machines. Thus we use examples such as untrusted processes, files and directories, trusted kernel system services, etc. The Z specification and ultimately the formal apparatus used to conduct the proofs are deliberately phrased in less emotive terms of alpha entities, gamma entities, etc, in order to avoid emphasising such connotations. While the specification undeniably can be used with such interpretations, it is not limited to such interpretations. In fact, the intended implementation strategy envisages the use of protection facilities at a much finer grain than the usual TCB process/file granularity and great effort has been expended, both in the abstract machine specification and the proof strategy, not to preclude such use.

2. Overview of Machine Structure, Functionality and Policy

2.1. Basic Structure and Functionality

We identify a system as consisting of a finite set of entities, where entities are any tangible aspect of the machine which are the subject of manipulation by the functionality of the system.

We consider entities to have attributes, size, colour, shape, etc, of which the primary attribute is that entities are typed. Thus entities may be files, directories, command line interpreters, IO devices, untrusted processes, system services, etc.

As well as having attributes, entities are considered to have relationships to other entities. Thus files may have a parent directory, sibling files, etc. A process executing a program may have access to an IO device and some directory or file. There is a concept that an entity may use some of its attributes to select from amongst a number of entities to which it is related. Thus, the process may use some of its data attributes, a name string, to select from the directory to which it has access, one of the files for further access. Having executed such an "open" command the process will now have a new relationship to a new file entity, an open file descriptor.

Using these relationships and entity type attributes, the command repertoire of the machine will define the possible structure of the machine, given some initial state structure.

Thus, it is not possible to give an exact picture of the machine but rather we can only demonstrate a gross pattern to which all instances of it will conform and to give a few representative examples of the effects of the commands. The Z specification of section 3 on the other hand precisely defines the machine solely by listing the attributes and relationships of the initial state and defining the repertoire of commands.

In this section we shall do this a little more accessibly by using a pictorial representation.

To capture the functionality and security characteristics of our archetype machine it is necessary for us to define six sorts of attribute and a single structural relationship. Of the six attribute sorts, four are used to frame the mechanisation of the security policy. Before examining these we shall first present the application functionality and the structure of the machine alone - in effect the insecure, commercial variant of our abstract machine.

2.1.1 The Basic Typed Entities

For this we need only the notion that entities have data attributes and are typed. The single structural relationship is a graph. This is not completely arbitrary but the constraints and structure can only be explained once the main entity types have been introduced.

Entities are typed into the following types.

- **Alpha** • Loosely these can be thought of as the persistent storage objects of a conventional hierarchical filing system, except that they cover both the non-terminal directories and the leaf files.
- **Trusted Path** • These represent the fixed IO devices of the system together with associated trustworthy software drivers.
- **User** • These represent processes running trusted command line interpreter code and which communicate with the human user via a trusted path. They are thus an accountable proxy to the human user.
- **Certificate** • These are used in mechanising aspects of the security policy which depend on separation of duty in the form of n-person rules. The certificates represent passive indications of the agreement of a person to some action and act in effect as signed forms of approval in the same way as in the real world certain procedures require "forms in triplicate with all the necessary signatures".
- **Gamma** • These represent processes running untrusted code.

The essential structure of the machine in terms of the persistent storage objects is then pictured as follows

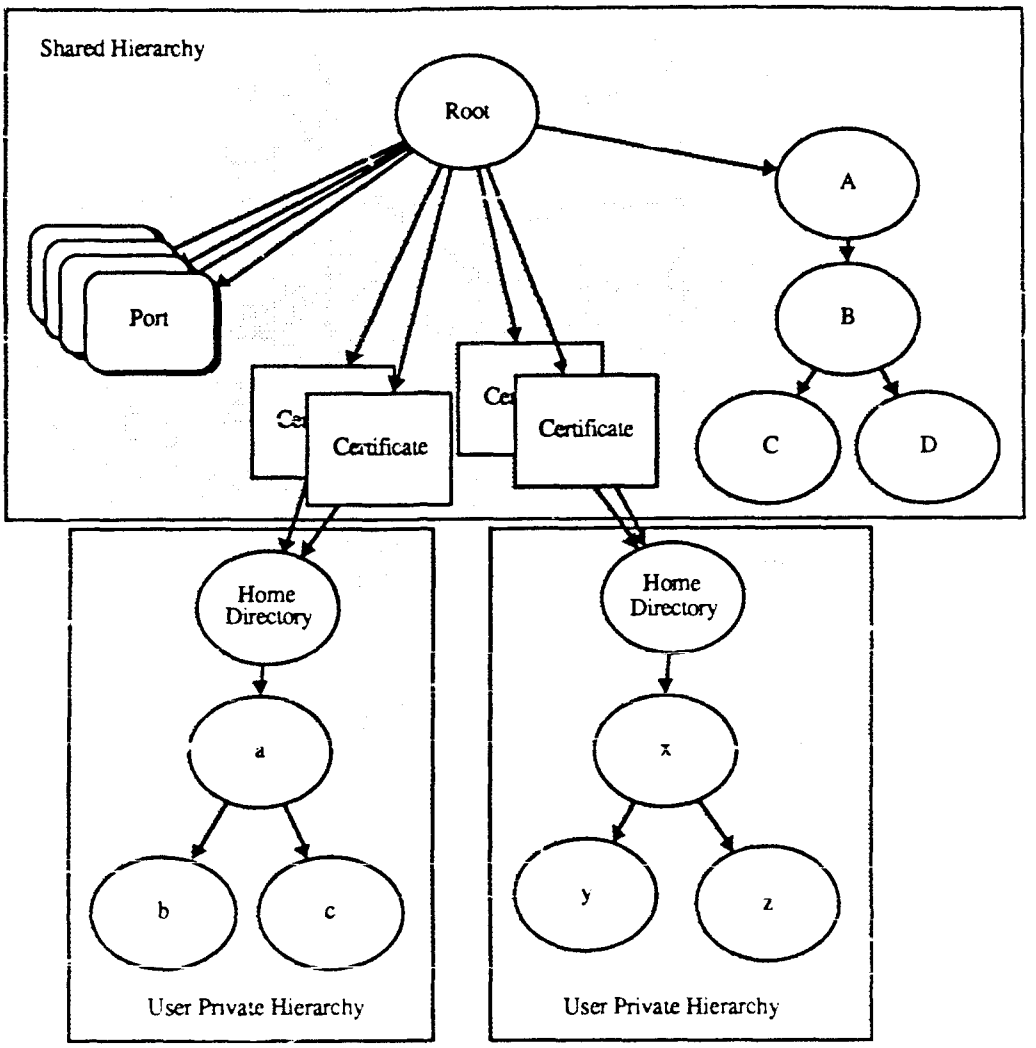


Figure 1.

Root, A, B, C, D, a, b, c, x, y, z, and the two home directories are all alpha entities. The certificates are stored in the alpha structure and serve in this case to separate the private storage areas of users, represented when not logged in by the home directories, from the common shared storage area represented here by the hierarchy A,B,C,D. The trusted path entities, labelled ports in the diagram, are also considered to be stored in the alpha structure as shown. They are shown drop-shadowed to distinguish them as potentially active entities which instigate commands and state transitions, as opposed to the passive nature of the other entities shown here as flat objects.

Root is a distinguished member of the alpha entities and is considered the starting point of what is essentially the hierarchy which represents the entity relationships. The commands of the machine ensure that the pure alpha structures from root, in this example A-D, and the pure alpha structures from the home directories, in this example a-c and x-z, are all single rooted hierarchies with no cross links.

The multiple certificates pointing at an object are the single exception to this rule. The certificates shown here are those required for an n-person authorisation of user logins, as we shall see later, this structure also occurs within the shared hierarchy when an alpha object is to be downgraded, again requiring n-person authorisation.

Login is modelled as a command instigated by a port entity which creates a user proxy entity connected to the port, root and the user's home directory thus,

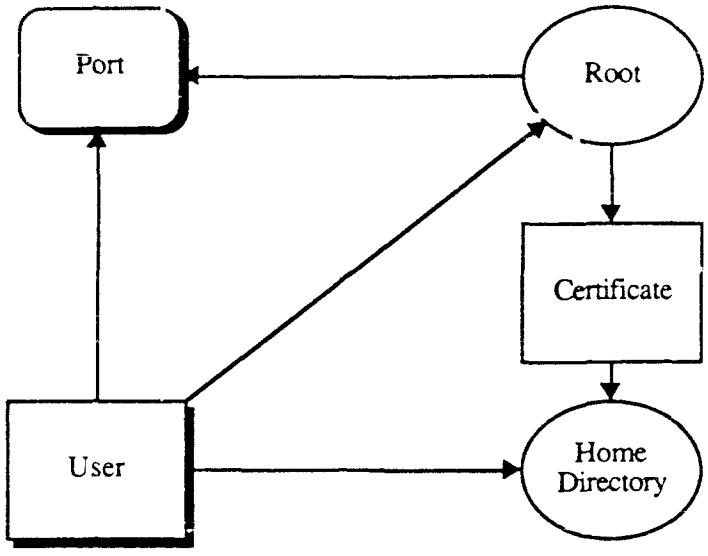


Figure 2.

Note that the user points into the existing structure and is therefore effectively a new root into what we like to think of as an essentially hierarchical structure. It is these effects which prevent us formally describing the overall entity relationship as a hierarchy and yet does not entail that the relationship is an arbitrary graph in practice.

Commands are available which enable such a user entity to traverse the alpha hierarchies from root and its home directory, and to modify these hierarchies by creating and deleting alpha entities. The user proxy is assumed to be a trojan horse free, command line interpreter like, entity and all of its actions are assumed to be accountable to the human user and initiated under the guidance of the human over the trusted path port. Such interaction is covered by commands which allow the user proxy to read and write the port to which it is connected. It is the only entity which is able to do this as only login creates such a connection. For simplicity we shall omit the port and these interactions from the diagrams leaving these assumptions implicit.

So after user activity the user proxy could end up gaining connections to other entities thus,

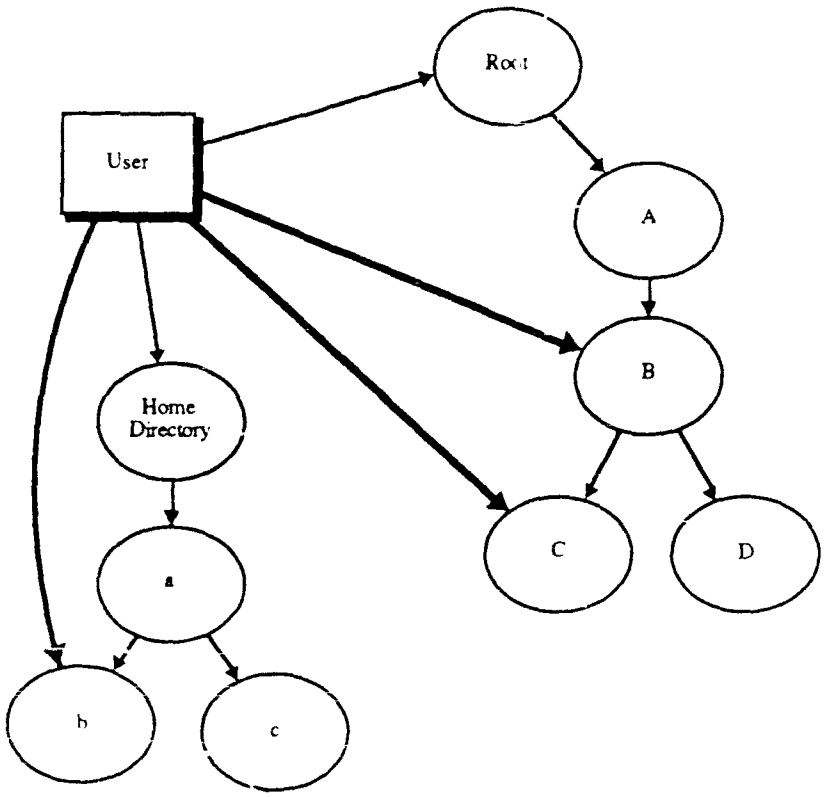


Figure 3.

Commands are also provided for the user entity to read and write alpha entities. In practice, for reasons we shall see when we consider the security policy repercussions on the functionality and structure of the machine, these commands are not expected to be widely used in the users' accomplishment of their daily tasks. They are presented for completeness and to allow a contrast to be shown between the functionality achievable with our abstract machine and policy and extant policies where these commands would effectively be the only ones available despite the extreme limitations put on them by the security policy.

In our approach users achieve their main tasks using untrusted software represented by gamma entities. User entities are therefore provided with a set of commands for initiating, interacting with, and terminating gamma entities. Gamma entities have available the same functionality as a user proxy in terms of interacting with the alpha storage structure, thus they can traverse, create, delete, read and write alpha entities. They do so however using a different set of commands which enforce different security constraints from those imposed on user proxy entities. In essence whereas the user is free to modify the structure but is limited in reading and writing data the converse is true for gamma entities. The constraints imposed by the security policy are such that they have only limited scope for creating/deleting alpha entities but are relatively free to read and write entities. It is this division which allows our approach to provide the flexible functionality required by users without compromising the assurance of security.

Of course gamma entities are totally debarred from executing the commands used by the user entities which utilise separation of duty, downgrade, certificate creation and manipulation, etc. We currently debar gamma entities from spawning further gamma entities though this is not thought to be an inherent requirement of our approach.

Gamma machines do not communicate directly with the human user over the trusted path port device. Their only access is to the user proxy which mediates such access. A gamma machine is in relation to its user parent in much the same way as the user is to the trusted path with the important exception that it is the user proxy parent which instigates exchanges with the gamma entity, ie the user reads and writes both the gamma entity and the trusted path port.

A user may control many gamma entities either synchronously or asynchronously and the environment of each gamma machine, in terms of its visibility of the shared and private hierarchies, is determined by the user proxy when the gamma entity is instigated. Obviously, this must be a subset of that available to the user at the time it instigates the gamma machine. Gamma machines are not constrained to work exclusively in the private or shared hierarchies but can interwork as in this example.

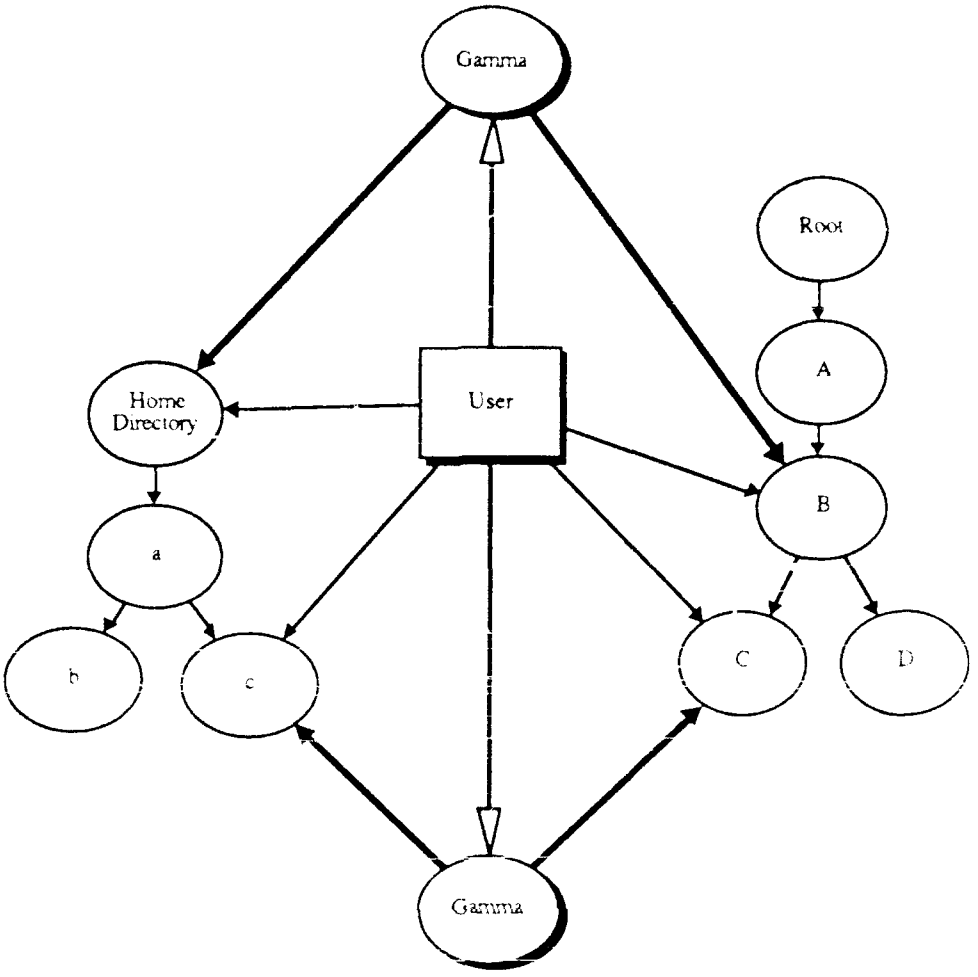


Figure 4.

Before considering the structural and functional aspects of creating and manipulating certificates we must review our policy assumptions as the interesting part of these transitions are the propagation and manipulation of the security attributes.

2.1.2. Labels, Roles and Certificates

The basic notions which underlie our policy models formulation of security can be paraphrased in the following examples in terms of entities and attributes.

When something happens on the machine, ie some aspect of the entities and attributes relationship or the structural entity relationship changes, then information can be said to have flowed from some entities of the machine to some others, from sources to destinations.

Most extant policies would simply insist that in order to maintain security the classification labels of the destinations should dominate the labels of the sources. Extant policy models vary in how completely they capture the notion of information flow. In our model, information flow is defined in true information theoretic terms, thus sources are identified not simply as those entities which are overtly observed in a transition but also those which only signal information indirectly. For example, the instigator of a third party copy command while not explicitly observed conveys information in its choice not only of which entities to copy from and to but simply by its choice of whether or not to even attempt the copy.

For the purposes of this overview we can consider such transitions of the system to consist of a single source entity and a single destination. Consider the following transition

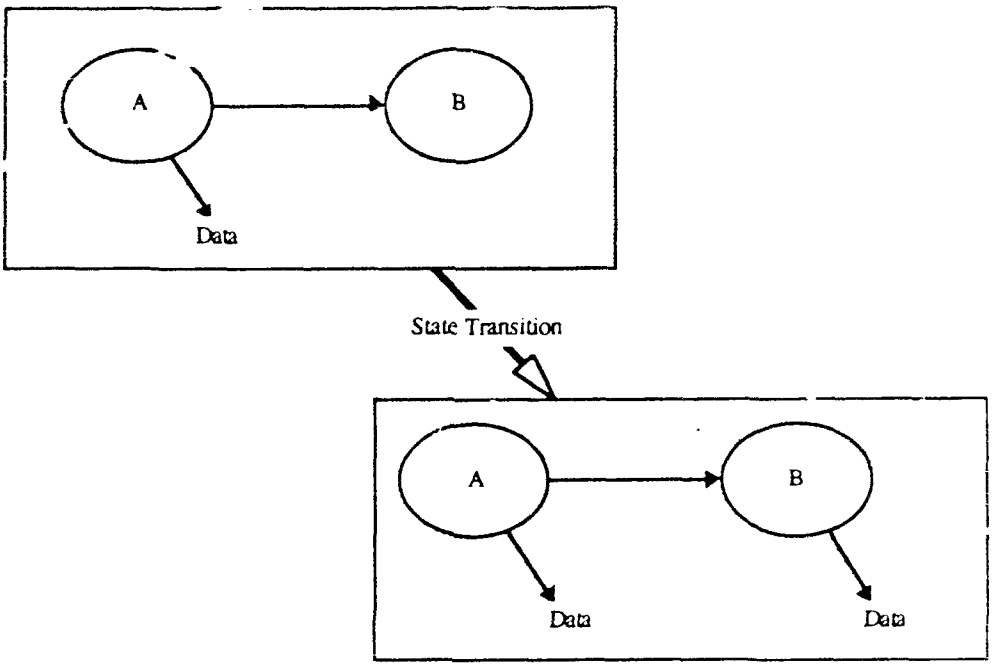


Figure 5.

which represents A as a source writing its data attribute into the destination B. As with any policy model, for the system to be able to execute a decision procedure on whether or not this is permissible, labels are required on the entities to reflect the sensitivity and handling requirements of the information held in that entity. Typically, extant polices would appear thus

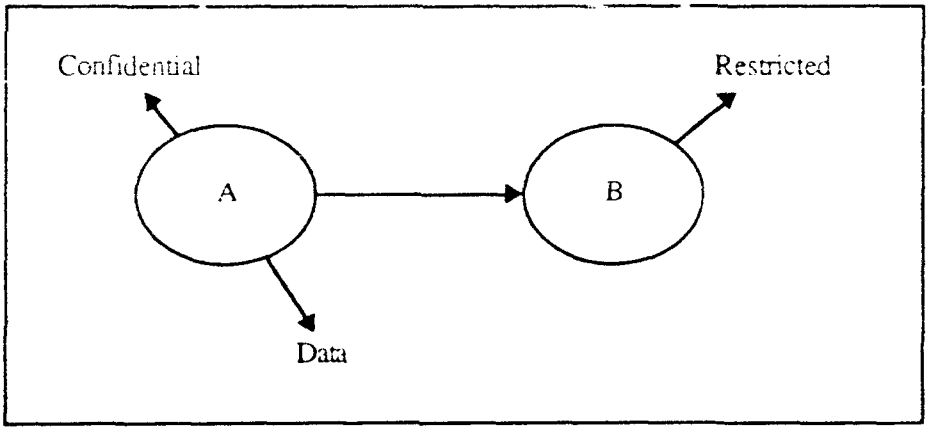


Figure 6.

In the state in which the transition was requested the source is not dominated by the destination so this transition would be disallowed. In our approach we would check if the transition was allowed by separation of duty. This requires additional labelling information about the individuals responsible for the disposition of information. Thus in our approach the entities would have labels as follows

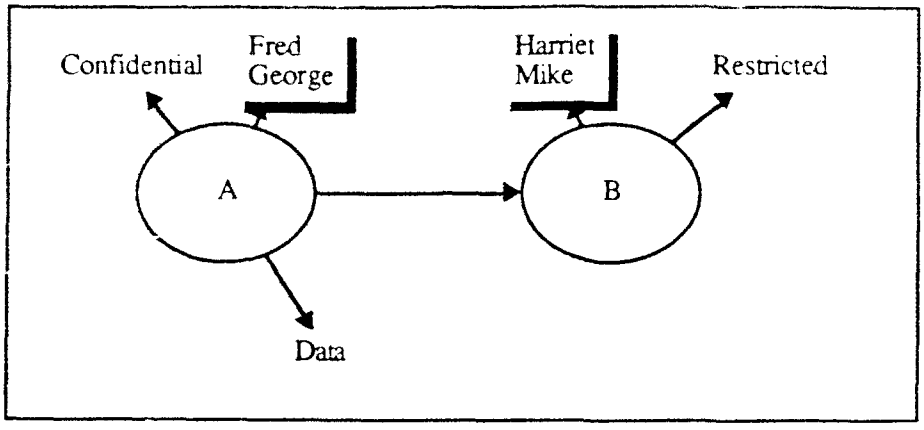


Figure 7.

Our motivation is that if an entity instigates a transition which moves information from some entity to another which does not dominate it then the instigating entity must show that the responsible individuals for the sources of that information agree to that transition. The individuals responsible for an entity's information content is signified by a conflict list. Fred and George and Harriet and Mike in the example. The agreement of these individuals is achieved with certificates. For the sake of this example we will assume that A is also the instigator of the transition. For the transition to occur certificates which concur with the transition and which are signed by appropriate individuals must be in the possession of the instigator. This is represented thus

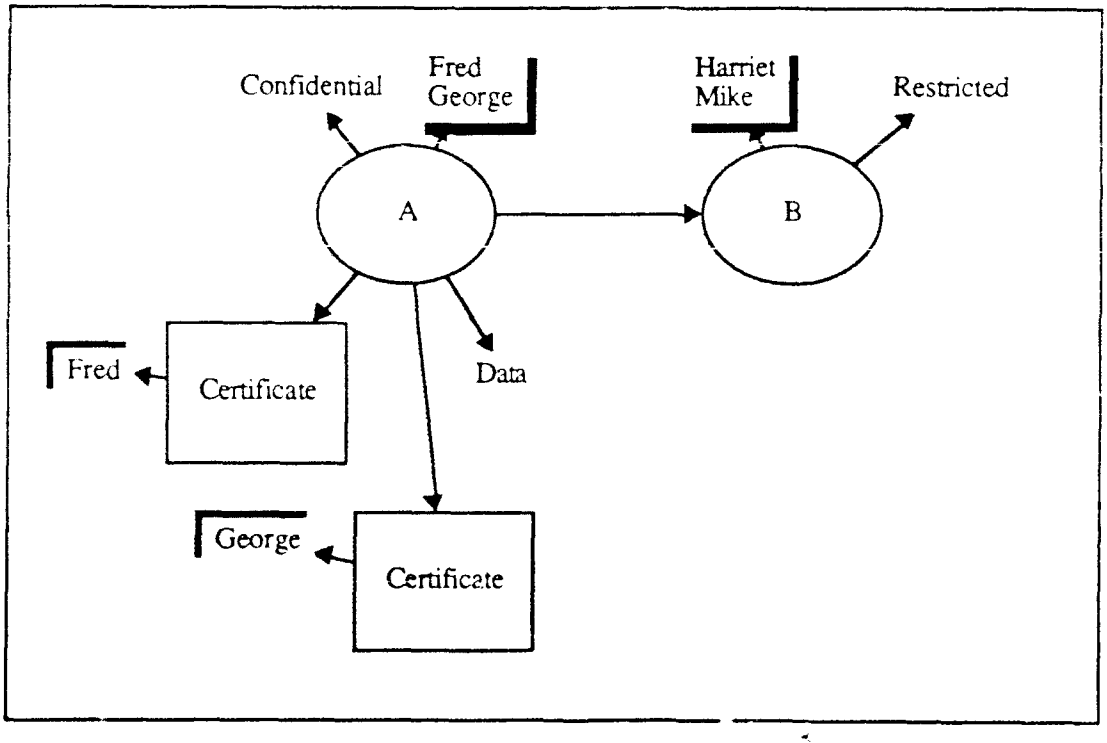


Figure 8.

The signature of a certificate is represented by a role or identity attribute.

We could extend this idea by saying that as well as having the agreement of Fred and George to release the information we also require the agreement of Harriet and Mike to assume responsibility for it. Apart from adding a little complexity this does not essentially change the nature of the mechanism or the notion of separation of duty security. Workshops on separation of duty notions have developed various classifications of such schemes, weak versus strong, dynamic versus static, etc. All of these are easily adopted into this basic mechanism. For the purposes of our research adding such syntactic complexity was not considered useful so the machine we have specified and proven uses the scheme outlined above.

In fact, even this simple scheme is actually more complex than pictured above As we have said our notion of information flow is complete and so in the transition pictured above not only is A a source, because it contributes data and is the instigator, but so too are the certificates. This is because their presence or absence and their concurrence with the transition requested affect the outcome of the transition. The certificates themselves also have intrinsic information content, itself protected by a classification label and a conflict list, thus unless care is taken inclusion of a certificate may incur a requirement for more certificates in order to approve the total information flow from the combined sources of the transition to the entity B.

Also the above outline does not indicate how certificates store the information as to the exact transition that they are intended to permit.

In our research we primarily wanted to establish the tractability of this approach so we have adopted the approach of using the minimum specification artefacts to approximate the functionality required. Thus we have tried to keep the number of attribute sorts and entity types to a minimum. Hence rather than allow downgrades, relative to the basic classification labels, to occur in the general transitions indicated above we have specified a single downgrade mechanism which enables us to keep the specification overhead to a minimum.

Before describing this, some further comments are in order on the interactions which occur between the standard lattice and the new conflict list/certificate roles mechanism. Consider the transition where information flows in accordance with the standard classification label lattice but where the destinations conflict list is not a superset of the sources conflict list. Specifically, consider that the destination has a singleton list consisting of the transition instigators identity or role. Such a transition obviously allows a downgrade to occur without separation of duty by the simple expedient of executing two steps - first move the data to somewhere where I can downgrade it on my own.

In reality therefore we do not have two security labels, where one set is only checked when a downgrade is attempted, but instead have a single label with an extended definition of dominates and allowed lattice flow. Thus the conflict list is really an extension of the standard lattice. A label dominates another if the classification component of the first dominates, in the classic sense, the classification of the second *and* the conflict list of the first is a superset of the conflict list of the second.

Our security policy is then that of no flows down in this extended lattice with separation of duty enforced by the conflict list subcomponent when this extended check fails.

So, we can now define two more of our attribute sorts used by the security policy in addition to the data and entity type attributes - the security label attribute, which is a structured attribute consisting of the classic hierarchy plus compartments sensitivity label and a conflict list of identities or roles, and the role/identity attributes, used to signify the accountability of an entity to a human user.

Thus our typical entity is considered to possess the following structure

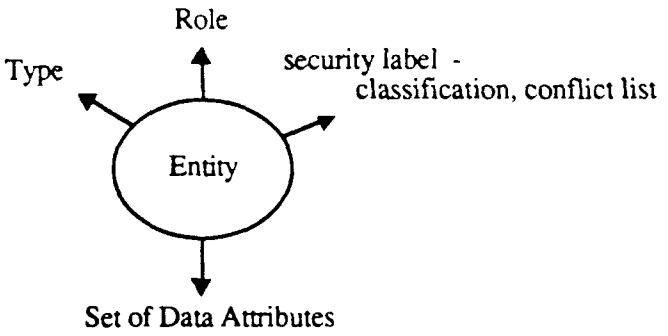


Figure 9.

The next major change in our policy approach is that the security labels of entities are not required to be fixed, labels can float in our machine. Thus returning to our modified example

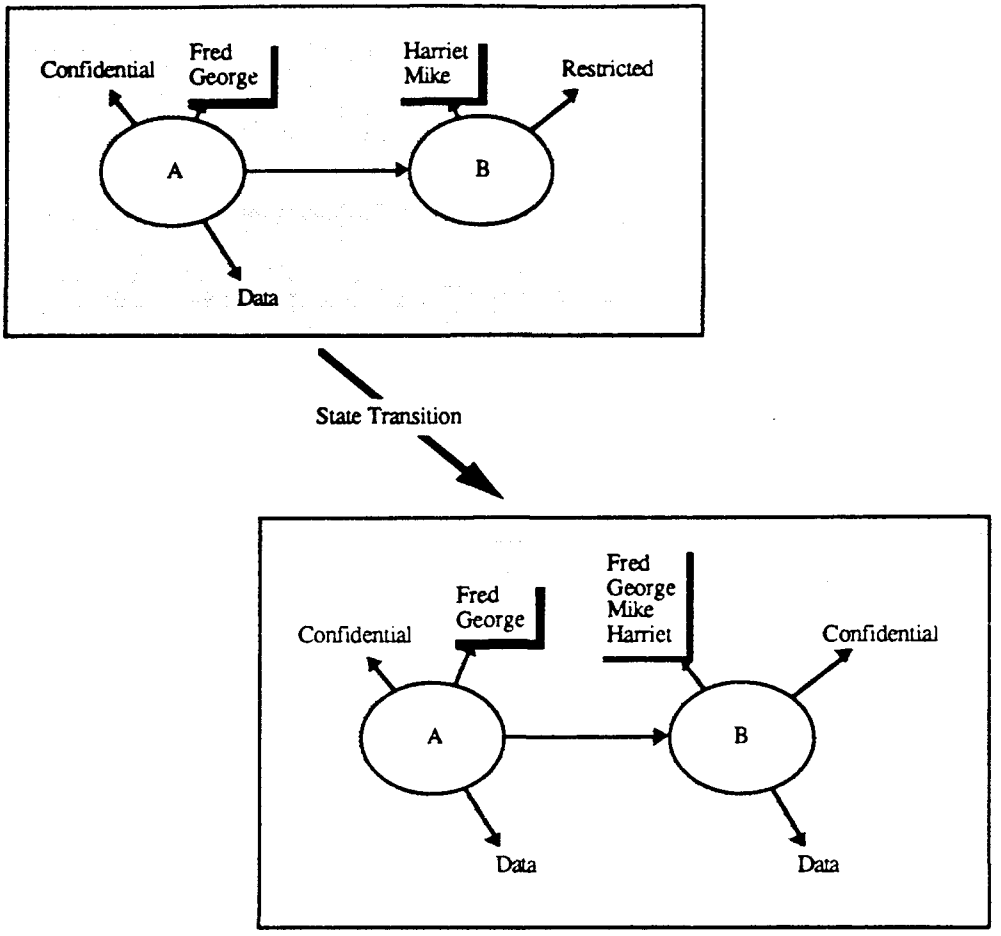


Figure 10.

The extended notion of "no flows down" can be enforced by simply requiring that the labels of B reflects its contents after the transition by simply floating the label to the least upper bound of A and B's security labels.

2.1.3. Inner Labels and Label Objects

Our notion of downgrade depends for its mechanisation on a further type of entity, a label object. The motivation for the function of these objects, and another attribute sort associated with them, inner labels, requires that we consider the formal basis of our policy model a little more closely.

Extant policies of information flow in a lattice are usually formulated in terms of systems with a fixed number of entities with fixed labels. In our machine we allow the "creation and deletion" of entities and as indicated above do not constrain labels to be fixed for the lifetime of an entity. These two relaxations introduce two further flows of information in addition to the overt flows of information due to reading and writing data in entities.

The first allows covert flows by means of modulating the availability of the "fact of existence" of an entity. In our model, as in real systems and most extant models, entities do not in fact appear and disappear into and out of thin air. Entities are usually allocated or activated from free lists, etc. In some extant models these are either not explicitly modelled, or, if they are modelled, are not covered by the policy statement of security. In our approach they are explicitly modelled and therefore captured by our complete information theoretic definition of information flow. In our model movement to and from a free list by a garbage collector or system service is reflected by the changes such a specification indicates in the structural relationship of entities.

The second relaxation allows covert flow by modulating the "permission to access" due to relative lattice labels imposed by the policy statement. Thus the classification of an entity, designed to protect the information content of the entity, itself has an information content requiring classification and protection.

The notion of an enforced, non-bypassable structural relationship amongst entities is imperative in order to be able to address these two issues. For fact of existence this can easily be seen in the example of labelled files and directories. To find the existence of a file I must be able to list its name in a directory or use an open command in the directory,

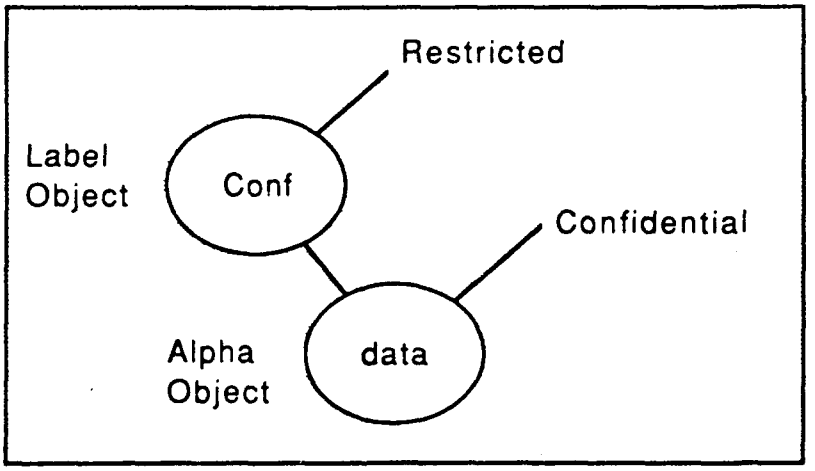
etc. If the classification of the directory is such that I cannot perform these operations then I cannot know of the existence of the file. This property quickly generalises to the hierarchy of labelled objects which we know and love in most extant models and implementations. Thus, newly created files are stored in directories whose label reflects the sensitivity of the fact of existence of the file. This of course is dependent on the hierarchy commands upholding the property that only suitably cleared entities can traverse the tree and enter such classified branches. In traditional file systems using user supplied names, this protection extends not only to the file interrogation commands but also the commands to create and name files. By probing with user supplied names the existence of files can be inferred by name clashes.

This mechanism can also address the fact of classification. If the fact that a file is classified confidential is restricted even though its existence is unclassified then it should not be stored in a directory below restricted.

With this approach a secret file stored in a restricted directory tells one that the fact of existence of the file is restricted, OR, the fact of its classification at secret is restricted, OR, both.

Our information flow definition, in order to be complete, would regard an attempted read of a classified entity by an uncleared entity, even though it is refused, as a flow of information from the classified entity to the uncleared entity. In otherwords executing a security check introduces a security flaw. To counter this it must be the case that one must dominate any entity against which one can execute a command. Those which one doesn't dominate must be such that one cannot execute any commands against them. This property is provided by our structural relationship, all commands reflect in their specification that the indication of entities is relative to the instigating entity. This, at first nonsensical, notion that one only has access to that which one is allowed access is mechanised by a notion of label objects.

Every alpha entity is viewed as really consisting of a pair of entities, a label object which points at the true alpha object. The label object can contain, as data, an indication of the classification of the alpha object to which it points. The label object's label indicates the sensitivity of the information it encodes in the normal way as for any other entity. An entity which dominates the label object's label can therefore read its data, the alpha object's sensitivity, and determine whether it would be granted access to the alpha object, a check which does not entail information flow from the alpha object.



Thus, this indirection allows the normal policy notions of entities dynamically gaining and losing access to objects, with a security check on such access, without there being an insecure information flow.

In this scenario the label of the label object classifies the classification of the alpha object. Unfortunately, the fact of existence of the label object, alpha object pair cannot be disambiguated from the fact of classification as the label object must be stored in an alpha entity which dominates its own label.

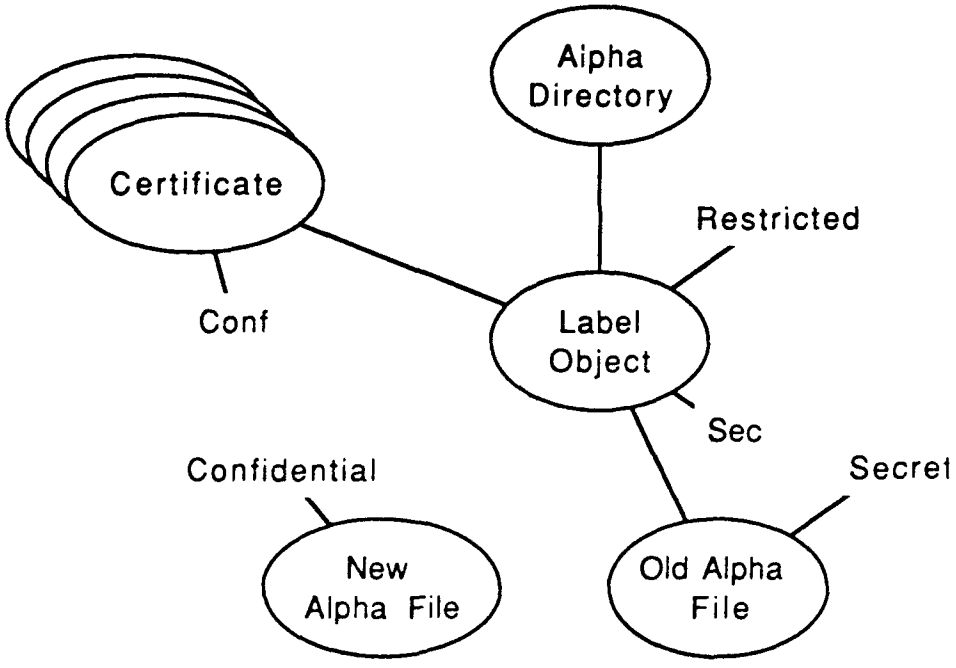
The label on a label object is not required to dominate the label of the alpha object to which it points. The strict "you must dominate what you point at" is only applicable to the active subjects of the policy model. The relationship amongst passive objects of the policy can be anything which the commands require in order to uphold the policy.

The label of an alpha entity stored as data within a label object is the motivation for our third security related attribute sort, inner labels. These simply reflect that security labels have a representation which is matched by some representations of data and allow us to coerce between the two types.

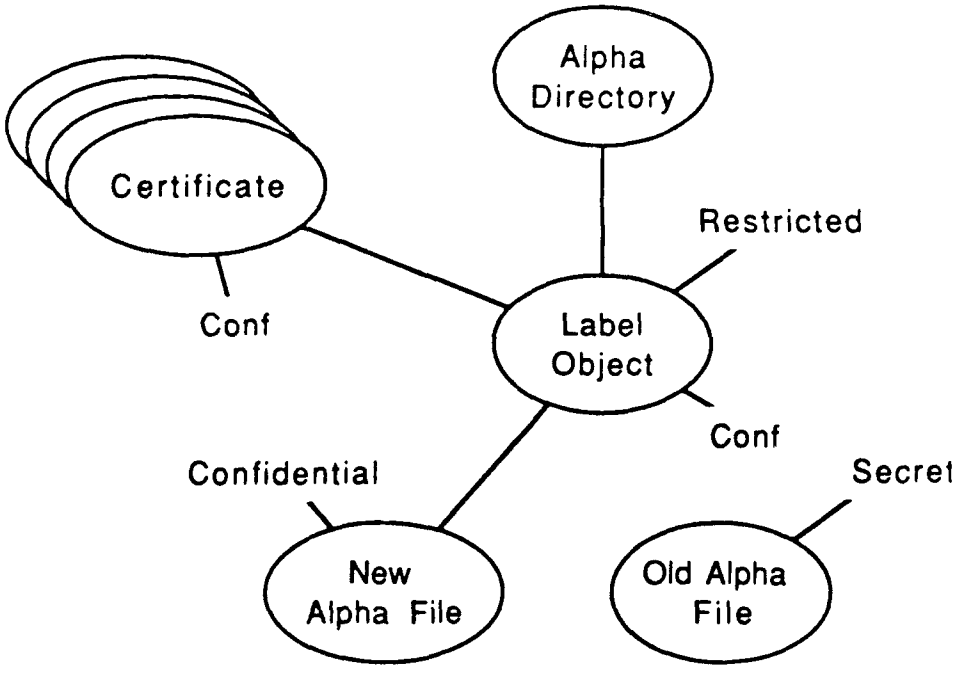
2.1.4. Certificates and Inner Roles

Finally, label objects also allow us to model the separation of duty notion of downgrade with the minimum specification overhead while at the same time retaining the intuitive functionality that users would require of such a transition.

A certificate is inserted into the hierarchy such that it points at the label object of the alpha object which is to be downgraded.



The certificate contains the data of the alpha object which it is agreed can be downgraded. This copy when the certificate is created accounts for "freezing" the data during the multi-step downgrade procedure preventing "Time of Check-Time of Use" (TOCTOU) attacks. It also contains, as inner labels, the new labels agreed for this data by the owner of the certificate. When sufficient certificates, which agree on the data and labels, are accumulated a new alpha object is created with those attributes and the label object is adjusted to point at this new alpha object.



This reflects the typical "name sameness" which users expect of downgraded documents. Each time they go to the registry and get document "X" they expect to get the latest document regardless of any downgrading that has occurred in the interim. In the downgrade command the label object is preserved in the directory thus all access to the file which are indirected via this automatically pick up the downgraded document.

The final attribute type of the policy model are the inner role attributes. These arise in the mechanisation of user registration when certificates are used to indicate the role and labels of permitted user login. The certificates role indicates the role of the user approving a registration. The role of the user which it is approving must be held as data within the certificate. Thus, in the same way as inner labels, inner role simply represents a coercion of data.

2.1.5. Join Entities

Finally, there is one last entity type to consider before our machine is complete. These are the "join" entities. These are associated with gamma entities and reflect an abstraction of a property of certain machine architectures and implementation strategies. The scenario being addressed is one where a user wishes to manipulate an untrusted data structure with one or more pieces of untrusted software. Depending on the architecture and chosen implementation of these entities there may or may not be the possibility of interaction directly between the untrusted entities. For example, if the untrusted software is run as processes there may be system introduced semaphores on their access to the shared data structure. Using these it may be possible for Trojan Horses to communicate in ways not anticipated. In such cases, if one of the processes opens a classified file and has its label floated to reflect this then to be safe the other processes "joined to it" by the semaphore should also have their labels floated on the assumption that the first process will signal the data to them. Thus, while in the model there is no direct access to account for their labels floating, we wish to indicate that certain entities should be treated as a group with their security control treated in concert as a single entity. This is the purpose of the join entity and the join command. The join entity simply records the members of the group by pointing at them. Every untrusted gamma entity is associated with a join entity. The join command takes two join entities and ensures that each points at the union of its own and the other's entities and the other join entity.

2.1.6. Complex Sources

We summarised our security policy as that of no flows down in an extended lattice with separation of duty enforced by the conflict list subcomponent of the lattice labels when this extended check fails.

In reality we allow a second condition under which the strict no flows down in the lattice is relaxed. The justification for this also stems from the real world considerations which give rise to separation of duty as a notion of security.

In the real world when a human user creates a new file in a registry, a new project is started, etc, other humans are aware of the existence of the new object. This "one bit signalling channel" via the existence of objects is unavoidable. It is not considered a threat in the human environment because humans have much more convenient channels than signalling through patterns of project creation, deletion, etc. We carry this notion forward into our machine environment. Obviously software written by a user, command scripts and other automated schemes, may attempt to use such channels, effectively because they are able to use the power of the machine against itself. However, a trojan horse free, trusted path interface to a human user, which carries out one for one events on behalf of user actions on that interface, can be assumed not to be utilising such channels. We refer to entities on the system carrying out certain actions which technically violate no flows down, but which are allowed by virtue of such threat analysis arguments, as "complex sources".

We like to think of this argument as simply a very weak form of separation of duty, the user will not use such a channel not because of the conflicting motivation of some other user but simply because he lacks the motivation himself, it is simply too arduous to try and signal information by creating and deleting files with all the consequent mouse actions, dragging icons to wastebaskets, confirming dialog boxes, etc.

Given this initial whirlwind overview to provide the motivation for the machine structure and policy as a whole we can now turn to examination of the individual aspects in slightly more detail, including an outline of the commands available.

2.2. The Persistent, Passive, Storage Structure

Ignoring for the moment certificates and trusted path entities, the persistent, passive storage structure of the machine is a hierarchy constructed from alpha and label objects.

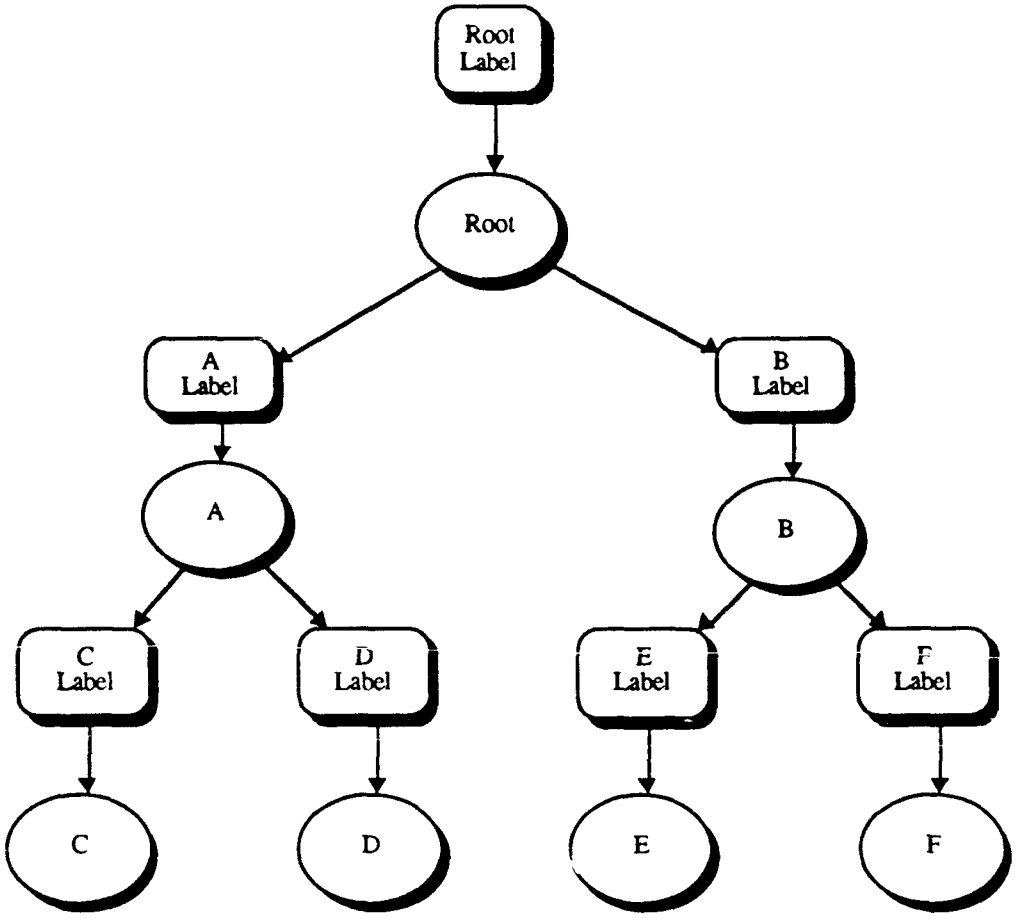


Figure 11.

All commands are such that they only work on objects to which the active entity has direct structural access. Given access to a label object entity, the user and gamma active entities of the machine can traverse the tree below that object using the following commands.

2.2.1. The Open and Examine Commands

The only command available on a label object is to "open" it. This checks the inner labels of the label object against the invoking entities security labels. If dominated the open command gives the invoker direct structural access to the alpha object. It is thus a property of the system that at all times the objects to which an active entity has direct access are always dominated by the active entity. The active entities clearance is its label in the case of a user entity and the label of the parent user in the case of a gamma entity. The open command in the case of a gamma entity also ensures that the gamma entity's label, as well as all its joined entities', floats to reflect the opened alpha object.

As well as being able to read and write data from an opened alpha object the user and gamma entities can "examine" the alpha objects related entities. Thus having opened root label, in the above example, examine would reveal the A and B label objects and allow selective update of the user or gamma entity's relations so that it has direct access to these labels. It can now repeat the open of this new label object to obtain access to A or B. With this alternating pattern of examine/open/examine/open a user or gamma machine can traverse the tree, at least as far as the classification checks of open allow.

2.2.2. The Read and Write Commands

For both user and gamma entities read allows the attributes of any alpha object directly accessible to them to be obtained. For write a no write down check is executed. For a user entity the check is against the users clearance. For a gamma entity the check is against the floating label (high water mark) of the gamma entity.

The severity of the former check, which is the one used in most extant models, is why we say that users typically achieve their work by using untrusted software, gamma entities. In extant systems a user reading the unclassified weather report and using the trusted path to send a memo about calling off the golf game to a colleague generates a report classified at its clearance. In our approach, as in reality, the user uses an untrusted editor process initiated with a

low high water mark. If the editor works without Trojan horse intervention, it will generate an unclassified memo. The user will certainly notice if the Trojan horse has grabbed any secrets and encoded them in the memo because the no write down check on the correctly maintained, trojan horse constraining, high water mark will prevent the user posting the memo in the intended unclassified mailbox object.

2.2.3. The Create and Delete Commands

User and gamma entities can create and delete label/alpha object pairs in the hierarchy subject to security constraints. In information theoretic terms inserting or removing the pair is equivalent to "writing" the alpha object at which the insertion/deletion occurs thus the security checks ought to be those of the corresponding write command for a user or gamma entity. This is the case for the gamma entity but in the case of the user a relaxation is made. As stated we believe a human user on a trusted path will not use signalling side effects, such as creation/deletion of entities in a shared hierarchy. Thus while we apply the write down check for overt writing, we don't trust users not to carryout wholesale copying, we do not carryout the write down check in the case of creation/deletion.

It might be argued that this is a weak form of separation of duty on at least two fronts. First, it is only allowed for a human on a trusted path, not for software written by humans for which the tediousness of such signalling is not a serious inhibition. Second, although it has not been explicitly mentioned as a security measure, we would ensure that all TCB operations are audited. Thus the creation/deletion of objects would be audited and users may have to justify a pattern of such activity recorded in a log to their superiors. This motivation is an example of separation of duty.

It must be admitted that without this relaxation the flexibility of the machine becomes somewhat restricted. We believe this division of labour between trusted and untrusted software is a realistic approach to automating security conscious applications. The actual manipulation of data by complex applications is carried out by untrusted software guarded by high water marks. The structure of the environment in which such software and data is stored and shared is controlled by trusted software held accountable to human users. Thus we let the TCB and the machine constrain the overt flows of information and let humans handle the signalling channels knowing that they are not equipped to exploit them.

In this regime the flexibility of the commands when used by a user as opposed to a gamma entity is the exact converse of the normal read/write. The gamma entities tend to get restricted by the fact that alpha entities' labels cannot float to the high water mark of the gamma entity and the no write down rule therefore restricts their scope for where they can create/delete alpha entities. Users on the other hand are virtually free to create/delete where they like, subject only to the fact that they might have to justify such audited activity at a later date.

In both cases entities are created with no overt data content. Subsequent read and writing of data is subject to the standard read write restrictions thus the labels chosen for the new entities govern who can read and write them. It is thus quite permissible for a gamma entity of say secret high water mark to create an unclassified entity in a secret directory. It will be prevented by its high water mark from writing data to the new object. Any secret data encoded by the fact of existence of the new object or by its selected classification is protected by the secret classification of the directory in which the object has been created.

2.3. The Transient, Active, Process Structure

We will describe the process structure in terms of an established user process. The establishment of such a process using separation of duty in registering and logging in are deferred for the moment.

A user entity has access to a trusted path port in order to interact with its human user, access to root in order to indirectly interact with other users by storing work in the alpha hierarchy, and access to a home directory in which to store work in preparation for sharing or which is to be private regardless of classification.

To assist with this work the user can instigate gamma entities which are untrusted active entities. To retain security control and yet give such entities the flexibility to achieve their tasks, gamma entities' labels can float from the level given when they are initiated by the user up to the user's clearance. The user can interact with the gamma entity to give parameters and receive results, and to mediate interaction with the human user on the trusted path device. The user entity must mediate to ensure that the display clearly indicates to the human user which input and output is with the trusted user entity and which with the untrusted gamma entities. This will involve a convention in the use of windows, labelling, etc.

This view of the typical lifetime activity of a user entity provides a convenient order in which to review the commands available. The commands to manipulate the storage hierarchy have already been covered above.

2.3.1. The Read and Write Path Commands

These are quite straightforward with no explicit security checks involved. The implicit security features are that a trusted path entity can be accessed by only one user entity which must leave the device in a purged state at the end of a session, ie the logout of the user. In reality the interaction between the display and input device and the command line

interpreter are quite complex, see for example the MaCHO Specification[1]. The maintenance of windows and their labels in a manner which allows users to clearly forsee the consequence of their actions and which prevent untrusted software spoofing users is clearly a design issue which is security relevant, however, this is really a question of designing an appropriate man machine interface and ensuring that it is correctly implemented rather than policy specific software.

For the purposes of the policy, and the security policy checks which must be embodied, the abstraction of this to the simple information theoretic read and write transitions is sufficient for completeness of our abstract machine.

2.3.2. The Create and Delete Gamma Commands

The only constraint on a gamma entity's initial labels are that they are dominated by the user's labels. As in the creation of alpha entities and label objects, in reality gamma entities are always created one for one with an associated join entity.

2.3.3. The Read and Write Gamma Commands

These commands are sufficient abstractions of both the initial priming of a gamma entity with parameters from the user creation call and to receive results before the gamma deletion command as well as covering any intermediate mediation by the user entity in the gamma entity's interaction with the human user on the trusted path.

2.3.4. The Join Command

The join command does not effect the structure or behaviour of the machine in application functionality terms. Instead it acts to modify the coverage of the security mechanisms. Thus, joined gamma entities continue to exercise their normal interactions with the alpha hierarchy and do not obtain any extra functionality in terms of enabled interaction with the joined gamma entities, etc. Instead all security relevant operations to float the labels of a gamma entity (and its join entity) now act to float the labels of all entities in the joined group of gamma entities. Thus the interaction between gamma machines is an assumed linkage not explicitly modelled.

2.4. User Registration/Deregistration

Because of the policies dependence on separation of duty user registration in this machine is unusual in the sense that it avoids as far as is possible dependence on the notion of a single "super user" style approach to the registration of users, their initially allocated clearances and roles, etc.

While not explicitly modelled, the approach used is dependent on authentication and identification processes which permit of mutually suspicious authentication, non-repudiation, etc. These are properties usually associated with, though not exclusive to, public key cryptography approaches. The significance of authentication and identification to separation of duty based approaches has been widely discussed, in for example the WIPCIS programme.

The approach simply recognises that if a human users are to be held accountable for the actions of their computer proxies then they must have the opportunity to veto or amend not only the parameters of their registration on the machine but fundamentally whether they consent to be involved with it in the first place. This is simply achieved by making the registration process dependent on the human user requesting registration as the first step.

2.4.1. The Registration Request Commands

The human users granted physical access to a port of the system can request registration by effectively signing an application form which gives their undeniable identity and describes the clearance, in terms of sensitivity and role responsibility, for which the user wishes to accept responsibility. Thus the user says "I am willing to be entrusted with up to secret information and wish my questionable actions (downgrades for example) to be supervised by my boss and Fred".

Existing users of the machine, who are relevant to the request in terms of accepting responsibility for the user's requested clearance, in the example the boss and Fred, can then choose whether or not to approve the user working under the requested constraints. They are not able to modify those conditions. Thus the user cannot find himself unexpectedly entrusted with top secrets, which he didn't want because for example the penalties for errors are unacceptable to him, nor barred from material which he wished to access and which was the only motivation for registering as a user of the machine.

---

1 "A Z Specification of the MaCHO Interface Editor", A.Wood, RSRE Memorandum 4247.

These steps are mechanised by the direct analogy of the real world model of signed certificates and forms.

2.4.2. The Certificate Commands

The registration command creates a home directory for the new user with the parameters and identity provided. A home directory is merely a passive storage area for the user and does not imply any ability for activity on the system. This home directory is the subject of a certificate which records the putative users positive endorsement of the parameters. Other users can only approve certificates by first inspecting them, (one can't approve things blind) and then "signing" a copy of the certificate indicating their concurrence.

For housekeeping purposes users can also delete certificates they have created. If the user deletes the certificate created by a registration request he effectively deregisters from the system and disables future logins.

2.5. User Activation/Deactivation

The home directory with its collection of associated certificates is the basis of the mechanisation of the login process.

A human user with physical access to a port can also attempt a login. As with the registration command the human user provides an undeniable identity and a putative clearance. If a home directory with those properties can be found, and if the necessary certificates approving that directory are present a user proxy process is created on the port with access to that home directory. Logout simply removes this proxy.

2.6. Downgrade

The mechanisation of the multiple step, separation of duty, downgrade command is similar to the user registration/login process. Users can request the downgrade of an alpha object. This creates a certificate with a copy of the data to be downgraded and the new labels which are requested. The certificates labels reflect the current alpha object classification as we cannot assume the downgrade will be approved. The identity of the certificate "signs" the requestors concurrence with the labelling of the data with the new labels. Other users responsible for the data, by virtue of the conflict list label of the alpha object, can inspect the certificates associated with the alpha object and, if they concur, create a copy of the certificate with their "signature".

When sufficient certificates which agree on the data and the labels to be accorded it are amassed a downgrade command against the alpha object will succeed. This involves creation of a new alpha object with the data and labels indicated by the certificate with modification of the old alpha objects label object to now point at the new alpha. Thus the path of open/examine commands by which an entity accesses the downgraded object is not disturbed.

3. The Z Specification of the Abstract Machine

The Z specification of the abstract machine indicated by the above overview is given in the appendix of this document. The specification is in RSRE's ZADOK dialect and was edited and type checked by those tools[2].

In this section we make only general comments on the form and structure of that specification.

The specification is in the typical Z idiom for a Finite State Machine specification.

Thus a state is defined as a schema giving the functions which provide the attributes and entity relationships of entities. A value for these functions is given for an initial state.

The commands of the state machine are defined in the Z operations idiom, that is, in terms of the effects of the operation on the state. The initial state and the state transition command implied by these operation specifications thus inductively defines the abstract machine.

The one aspect of the specification's structure and form which is not apparent from the section 2 overview is the question of modelling creation and deletion of entities.

---

2 Hence, this document's source is held in Microsoft Word 4.0 format on Apple Macs while the appendix is held in RSRE's proprietary Flex format of the ZADOK tools on ICL Perqs. (All trademark and copyrights acknowledged)

In a typical Z specification this would be simply handled with an approach which identifies a "new" entity as simply one which was not currently involved in the state functions. In this specification a somewhat more explicit formulation has been presented and some explanation is required for this.

The formulation is present not from a desire to convey any particular implementation strategy. Indeed in striving for abstraction in order to avoid this the functionality is somewhat obscured. The motivation for the approach presented stems instead from a desire to emphasise a point which arises from the security policy modelling approach.

Creation and deletion of objects, even if implemented as allocation and deallocation of a finite population from free lists, etc, creates an unavoidable signalling channel through the modulation of resource exhaustion. The formal definition of information flow adopted in our approach is a complete definition in information theoretic terms. Therefore even when the exact mechanisms of allocation and deallocation are omitted from an abstract specification our policy approach captures this potential information flow which is fundamentally unavoidable[3].

If the normal Z idiom were to be used for modelling the system, when viewed from this formal flow point of view, any transition involving allocation of entities would constitute a flow of information from *every* entity in the system. In practice in the implementation the theoretical flow, while it cannot be eliminated, can be rendered impractical to use. Typically the allocation is by some kernel system service thus the flow from every entity in the system is to this kernel service. On subsequently allocating an object to a user of the service this information flow is not necessarily propagated directly. Consider an object released on to a free list by an entity X. Allocation from the free list might be from the head of the list and not the actual object released by X. Thus direct, usable, signalling from X to the caller of the allocation service is obscured by indeterminate length of the free list. Emptying a free list to remove this noise, ie to act on the resource exhaustion condition, can be prevented by least privilege quota's, etc.

The model therefore includes an entity which fulfils this signalling suppression role. It is presented in the specification as a "garbage collector". The specification is not of any practical collection algorithm thus this misnomer can be read simply as that part of the system intimately related to the allocation/deallocation transitions.

In the formal model this garbage collector entity can be presented as a "complex source". By definition in the model such entities are assumed to uphold the no flows down condition by means *ex machina*.

Thus the purpose of the garbage collector in the specification is not to provide a solution to this problem but to draw attention to this problem area and avoid conveying the impression that the problems of allocation/deallocation somehow "fall out in the wash" with our approach.

---

3 In a system where the formal structured system of interacting entities shares the finite resource of the machine dynamically. If the machine resources are pre-allocated into isolated divisions with no sharing of unused resources between such regimes then the situation does not arise. In effect the physical machine is not shared but partitioned into separate physical machines sharing only the power supply!

Appendix A

This appendix contains the Z specification of the Abstract Machine. The presentation of the machine is as follows.

First, the basic types for representing entities and attributes are defined. Supporting definitions, such as the lattice formed by security labels, etc, are also given.

Using these types the state of the abstract machine is then defined. An initial value for this state is then defined.

The commands or operations of the state machine are then described. These are introduced in a different order from the overview in the document and are more in line with describing how a fully functional machine is boot-strapped up from the initial state.

Thus the login/logout available to the system owner in the initial state are described. The ability of further individuals to register as users and the approval of such users is then described. This involves the traversal of the alpha hierarchy with open and examine commands as well as the basic certificate handling commands.

With the ability to create a population of users now shown, the basic alpha hierarchy manipulation commands available to them are now described followed by the commands available to them to create and delete gamma entities.

This leads to the description of the variations of the user commands available to the gamma entities followed by the interactions between active user and gamma entities.

The interaction between user entities and the trusted path are then described which provides the only input/output ability of the machine.

Finally, the downgrade commands are described.

The Basic Types

We parachute in the four following types (sets). E is the finite set of entities which comprise the system. Classification is the finite set of values which are used to construct the normal security levels lattice. Identity is the finite set of values used as names of people/roles and used to create the other notion of security levels used in SMITE, conflict lists. Datum is the finite set of values representing information encoded as data.

$$[E, CLASSIFICATION, IDENTITY, DATUM]$$

Entities are typed by a state dependent function into one of the explict types defined by entity_type.

$$ENTITY\_TYPE ::= Label\_Object \mid Join \mid Alpha \mid Trusted\_Path \mid User \mid Gamma \mid Certificate$$

Trusted Path entities are the IO devices of the system. Users are the trusted command line interpreter entities which act as proxies for the human users notionally attached to the IO devices. Alpha entities form the hierachical storage of the system which User entities can read and write, create and delete, etc, in response to commands read from the Trusted Paths. Gamma entities are the untrusted processes spawned by User entities and given subsets of the Users access to the Alpha structure.

In practice the Alpha structure includes Label Objects. There is a one for one correspondence between Alpha objects and label objects. The label object simply holds, as data, the classification and conflict list labels of the corresponding alpha object. The label objects provide two main functions. One, they allow the classification of an entity to be different from the classification of its classification, which is sometimes required. Two, they allow our analysis technique to work in the sense that they allow denial of access due to security checks to be analysed without assuming that there is a flow from that entity. In addition they also allow a nice mechanisation of downgrade.

Join objects are associated with gamma machines and represent the binding together of entities which share a common high water mark.

Finally, certificates are a variety of Alpha object used to mechanise the separation of duty transitions for registering users and downgrading alpha objects.

The following datatype re-definitions of the sets already introduced are simply a nice Z idiom for identifying some named distinguished members for use throughout the remainder of the specification.

$$CLASSIFICATION ::= Top \mid Bottom \mid classes \langle\langle CLASSIFICATION \rangle\rangle$$
$$IDENTITY ::= SysOwner \mid ids \langle\langle IDENTITY \rangle\rangle$$
$$E ::= RT \mid RTL \mid GC \mid GCL \mid entities \langle\langle E \rangle\rangle$$

Top and Bottom are the requirements for a finite lattice.

SysOwner is the identity of the system owner required in the initial state to allow the system to boot up.

RT is the root of the alpha structure, RTL is the label object for RT. GC is the garbage collector modelled as a user entity present from the initial state. GCL is a label object for the GC. The importance of the garbage collector will become apparent as we progress!

The following keeps the above types and members for use in the remainder of the spec.

$$basic\_types\ keeps\ E, CLASSIFICATION, IDENTITY, DATUM, ENTITY\_TYPE, Label\_Object, Join, Alpha, Trusted\_Path, User, Gamma, Certificate, Top, Bottom, SysOwner, RT, RTL, GC, GCL$$

basic_types :Module

Security Labels

This section describes the notion of security labels to be used throughout the rest of the specification. The policy model effectively uses two lattices.

First is the normal secret, confidential, etc, (including compartments) which is not modelled explicitly herein as it is all too well known. The second is the notion of lists (modelled herein as sets) of identities of individuals responsible for the information in an entity. These conflict lists are used to enforce the separation of duties controls when we step outside the bounds of the first lattice.

Because the vast majority of transitions, while not explicitly involving separation of duties, nonetheless, require propagation of a "no flows down" property in both lattices, it is convenient to handle both labels as a single security labels schema.

SECURITY_LABELS

$class : CLASSIFICATION$

$conflict : \mathbb{P}\, IDENTITY$

In these values we now define a top, bottom, dominates, lub, glb etc in the normal way to form our policy model lattice. The top security label ($\top$) has the Top class and all possible identities as its conflict list. The bottom label ($\bot$) has the Bottom class and the empty conflict list.

$\top : SECURITY\_LABELS$

$\bot : SECURITY\_LABELS$

$(\top).class = Top$

$(\top).conflict = IDENTITY$

$(\bot).class = Bottom$

$(\bot).conflict = \varnothing$

Dominates is defined as a partial order to form a lattice in the normal way. We indicate that dominates in conflict list terms is simply the superset notion (subset in schema since the type checker library does not define superscript⁻¹)

$\_dominates\_ : SECURITY\_LABELS \leftrightarrow SECURITY\_LABELS$

$\forall x : SECURITY\_LABELS \bullet \top\ dominates\ x$

$x\ dominates\ \bot$

$\forall x, y, z : SECURITY\_LABELS \bullet x\ dominates\ x$

$(x\ dominates\ y \land y\ dominates\ x) \Rightarrow x = y$

$(x\ dominates\ y \land y\ dominates\ z) \Rightarrow x\ dominates\ z$

$\forall x, y : SECURITY\_LABELS \bullet x\ dominates\ y \Rightarrow y.conflict \subseteq x.conflict$

Lub and glb are defined in the normal manner.

$lub\_ : \mathbb{F}\, SECURITY\_LABELS \nrightarrow SECURITY\_LABELS$

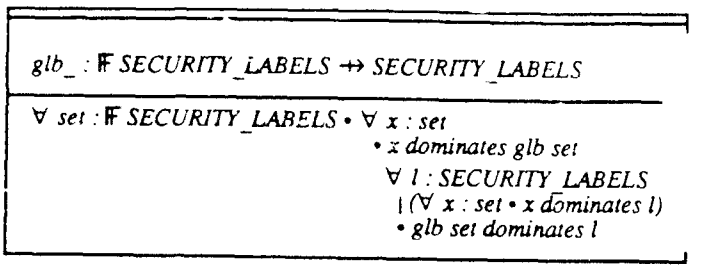
$\forall set : \mathbb{F}\, SECURITY\_LABELS \bullet \forall x : set \bullet$

$lub\ set\ dominates\ x$

$\forall l : SECURITY\_LABELS$

$| (\forall x : set \bullet l\ dominates\ x)$

$\bullet l\ dominates\ lub\ set$



labels keeps SECURITY_LABELS, ⊤, ⊥, *dominates*, *lub_*, *glb_*

basic_types :Module | labels :Module

The Definition of the FSM State

The state of the system is defined in terms of five total functions and two relations.

STATE

$g : E \leftrightarrow E$
$outer\_labels : E \rightarrow SECURITY\_LABELS$
$entitytype : E \rightarrow ENTITY\_TYPE$
$role : E \rightarrow IDENTITY$

$data : E \leftrightarrow DATUM$
$inner\_labels : E \rightarrow SECURITY\_LABELS$
$inner\_role : E \rightarrow IDENTITY$

The first four are considered notional "controls" on an entity which contains arbitrary data. Because of the requirement for entities to create and modify controls on other entities there is a need to coerce data to controls and vice versa. Rather than model this, necessitating rather more structure in data, we use a specification artefact of two other functions whose data nature is intimated by the prefix inner. This mimics the approach we take when using m-Eves to analyse the model. In practice, the proofs of the policies notion of information flow do not distinguish between controls and data, they are simply concerned with the structure and values of the state. This artefact does not weaken the proof of security and yet keeps the specifications relatively simple.

G gives the structure of the system. An entity scheduled as the instigator of a transition can nominate other entities as being involved only if they are in the range of its domain of the g relation. All of the state transition specifications enforce this.

Outer_labels gives the security label of an entity which always dominates the information encoded by that entities domain of the state.

Entitytype is the function which types entities. It is the command specifications' use of this in conjunction with g which gives a machine "structure" and which enables security to be achieved in the face of high functionality by using data hiding, encapsulation, etc; all of the ...ilities which one expects from a notion of data typing and as implied by the name SMITE.

Role gives the identity of the person ultimately responsible for the entity. It conveys security significance in terms of segregation of duties only for User and Certificate entities. For all other entities it may be of use in application terms, thus for a gamma entity one knows who caused it to be spawned, for an Alpha entity, who created it, etc.

Data, other than that it changes during reads and writes, etc, has no security significance placed upon it. Inner labels and inner role are only indirectly security significant for label objects, joins, and certificates. In all entities they represent simply a coerced form of data.

State Transitions

State transitions are defined thus:

ΔSTATE

STATE
STATE'
instigator : E

$entitytype(instigator) \in \{ User, Gamma, Trusted\_Path \}$
$\{GC\} \subset dom\ (g \rhd \{instigator\})$

This is intended to capture that a single entity is scheduled to initiate a state transition. All transition specifications observe the convention that the state values used are derived from the instigator or transitively from entities in its g. The distinguished entities RT, GC, and their labels are assumed available to all transitions, thus giving the TCB access to most of the system through g if required. Only a few of the operations make use of this feature. The scheduler is not modelled but is assumed to only schedule the sorts of entity we expect to be active and only then if they are not deallocated and on the garbage collectors free list awaiting sanitisation and reallocation.

The Basic "Untrusted" Transition

The following transition allows only the instigators data to be modified and covers most untrusted processing. This schema is used with renaming of RESULT as the base case of all transitions which may give an error response.

$\Phi STATE$
$\Delta STATE$
$RESULT : DATUM$

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$

$\{ instigator \} \lhd data' = \{ instigator \} \lhd data$
$data' \subseteq data \cup \{ instigator \mapsto RESULT \}$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

Here we see the basic form which all the state transtion specifications will take where the changes in each state function and relation are given in equational form for the before and after state.

basic_state keeps STATE, ΔSTATE, ΦSTATE

basic_types :Module | labels :Module | basic_state :Module

The Initial State of the FSM

The initial state of a viable system requires eight entities plus at least one labelled, trusted path, IO device.

Four of the eight entities are the root (RT), the garbage collector (GC), and their associated label objects (RTL, GCL).

The purpose of the other four will become clear when the methods of registering users and logging in are explained. They are the system owners home directory in the alpha structure (oh), a certificate authorising the use of that directory (oc), and their associated label objects (ohl, ocl).

This initial set of entities is described below.

Init_Entities

$$oh, ohl : E$$
$$oc, ocl : E$$
$$ports, ports\_labels : \mathbb{P}\, E$$

$$\#ports = \#ports\_labels$$
$$\#ports \geq 1$$
$$\#(\{RT, RTL, GC, GCL, oh, ohl, oc, ocl\} \cup ports\_labels \cup ports) = 2 * \#ports + 8$$

The initial structure of these entities is given below. Each label object points at its associated labelled object. Thus, ignoring this detail, the structure is essentially that root points at the ports, the garbage collector and the owners certificate. The owners certificate points at the owners home directory which in turn points at root.

The circle formed from root to a users home directory and back is what stops the garbage collector erasing the system. Thus last one out turns off the lights.

The garbage collector points at every one, and always will do, otherwise he can't be a garbage collector. The ports point at no-one and never will. There are no other connections defined by the initial g.

Init_Structure

STATE

Init_Entities

$$rng(\{RTL\} \lhd g) = \{RT\}$$
$$rng(\{RT\} \lhd g) = \{GCL, ocl\} \cup ports\_labels$$
$$rng(\{GCL\} \lhd g) = \{GC\}$$
$$rng(\{GC\} \lhd g) = E$$
$$rng(\{ocl\} \lhd g) = \{oc\}$$
$$rng(\{oc\} \lhd g) = \{ohl\}$$
$$rng(\{ohl\} \lhd g) = \{oh\}$$
$$rng(\{oh\} \lhd g) = \{RTL\}$$
$$g \rhd ports = ports\_labels \lhd g$$
$$\#(g \rhd ports) = \#ports$$
$$g(|ports|) = \emptyset$$
$$(\{RT\} \lhd g, \{RTL\} \lhd g, \{GCL\} \lhd g, \{GC\} \lhd g, \{oh\} \lhd g, \{ohl\} \lhd g, \{oc\} \lhd g, \{ocl\} \lhd g, ports\_labels \lhd g) \text{ partition } g$$

The types of the initial entities is much as one would expect from the names and roles of the entities described above. Root and the owners home directory are Alpha objects, the garbage collector is a User process, the owners certificate is a Certificate, the port is a trusted path, the label objects are label objects, etc. The entitytype of all the other entities, in the g of the garbage collector, are unspecified. Before they can play any part in the system they must be allocated from the garbage collector's "free list" at which point they will be sanitised and assigned their appropriate type, etc.

Init_Entity_Types
Init_Structure

$entitytype(RT) = Alpha$
$entitytype(RTL) = Label\_Object$
$entitytype(GCL) = Label\_Object$
$entitytype(GC) = User$
$entitytype(ohl) = Label\_Object$
$entitytype(ocl) = Label\_Object$
$entitytype(oc) = Certificate$
$entitytype(oh) = Alpha$
$\forall e : ports \bullet entitytype(e) = Trusted\_Path$
$\forall e : ports\_labels \bullet entitytype(e) = Label\_Object$

The only significant role assignments in the initial state is that of the owner's home and owner's certificate which must be equal and that of the system owner. As the roles of all other entities are not significant for simplicity below we show all entities as being assigned SysOwner role in the initial state.

Init_Roles
Init_Structure

$\forall e : E \bullet role(e) = SysOwner$

For outer labels the initial state mandates that all entities are labelled bottom, except for the garbage collector and the owners home directory. The garbage collector must be cleared to top as it observes all entities during its activity. The system owner is given the clearance Bottom and a conflict list of himself.

Init_Outer_Labels
Init_Structure

$\forall e . E \mid e \neq GC \wedge e \neq oh \bullet outer\_labels(e) = \bot$
$(outer\_labels(oh)).conflict = \{SysOwner\}$
$(outer\_labels(oh)).class = Bottom$
$outer\_labels(GC) = \top$

Similarly for inner labels the initial state mandates that all entities are labelled bottom, except for the garbage collector's and the owner's home directory labels which reflect the outer labels of their associated object and the owners certificate which gives the approved clearance of the owner, ie the outer labels of the owners home.

Init_Inner_Labels
Init_Structure

$\forall e : E \mid e \neq GCL \wedge e \neq ohl \wedge e \neq oc \bullet inner\_labels(e) = \bot$
$inner\_labels(oc) = outer\_labels(oh)$
$inner\_labels(GCL) = outer\_labels(GC)$
$inner\_labels(ohl) = outer\_labels(oh)$

In the initial state the only significant inner role is that of the owners certificate which signifies the permitted role of the owner, ie the outer role of the home directory.

Init_Inner_Role
Init_Structure

$inner\_role(oc) = role(oh)$

Thus, the initial state is defined by the conjunction of the above schemas.

Init_State

STATE
Init_Entities
Init_Structure
Init_Entity_Types
Init_Roles
Init_Outer_Labels
Init_Inner_Labels
Init_Inner_Role

initial_state keeps Init_State

Overview of the Commands of the FSM

At this point the most logical way of introducing the functionality of the abstract machine is as follows. First, to show how the system owner can log in and logout, then show how a new user registers, then how the system owner can login, move through the alpha hierarchy, review the request, and approve it.

Once we can see how a population of users with conflict of interests can be established in this way we can then look at the users basic manipulations of the alpha hierarchy, create, delete, read, write, traverse, etc. We can then examine the task of spawning and deleting untrusted software shells working on behalf of the user and examine their basic functionality with respect to manipulation of the alpha hierarchy. Here we will see the slight restriction in their functionality and the use of high water marks to track their activity.

We then look at how the user shells interact with the gamma machines and the trusted path.

Finally, we can examine the use of certificates and label objects in the alpha hierarchy to enable users to downgrade alpha objects.

The "Garbage Collector" Model

Before doing this however we must briefly digress into a discussion of the manner in which some of these operations are modelled, specifically the role of the garbage collector.

The operations fall into two distinct groups, those that involve allocation of entities and those that don't. By allocation we mean the sanitisation of an entity solely in the g of the garbage collector with the values it requires and linking it into some other part of g. All such operation are modelled as two distinct transitions, the first involving the obvious instigator of the transition where various checks are made and a second with the garbage collector as the instigator which simply allocates the necessary entities and initialises their values from its own state variables, set in the first transition. It is probably not immediately clear why we model the operations in this cumbersome manner, while at the same time trying to avoid detailed specification of the data strutures of the garbage collector.

The answer is that we wish to show the limitations of our proof analysis technique which requires such careful attention to detail. The operation of allocating an entity will always show a flow of information from every entity in the system to the entity doing the allocation. This is basically the resource exhaustion covert channel. If we modelled allocation operations as a single transition every instigator would receive information from all classifications and would need to be designated a complex source, which by definition does not propagate or utilise such signaled information. In the formal model and proofs, to obtain the flexibility of the abstract machine without such drastic measures, we indeed have to model these detailed two phase operations. In this Z DTLS style description of the abstract machine, we primarily wish to express the functionality of the machine and do not wish to obscure this with unnecessary detail from the formal proofs. However, we do not wish to mislead readers with a sense that allocation of entities some how "falls out in the wash" with our approach. The style of modelling is therefore a compromise between showing the functionality as straightforwardly as possible and drawing attention to the security obligations.

We begin our tour of the Abstract Machines functionality with the system owner logging in. This requires the allocation of a User "command line interpreter" shell, and thus we meet head on the need for the two phase transitions involving the garbage collector. With the above background this is hopefully not now confusing.

The Schema Structure of the Command Specifications

Before discussing login a brief note on the general form of presentation used for the operations throughout this document. Each operation is defined with four Z phrases.

Starting at the end of the description, each operation is defined as a bad case, where only an error indication is returned to the instigator, overridden by the good case. The bad case is defined as the $\Theta STATE$ schema with RESULT renamed to some command specific value, "LOGIN_NOT_OK" for example.

The good case usually consists of three schemas. The first, Θ<*op-name*>, introduces the personae dramatis in its signature. The predicate defines in terms of these, all the parts of the state which don't change during the transition. This predicate can therefore usually be safely skipped on first reading and is mainly useful for confirming various questions during more rigorous examination of the behaviour of a transition.

The second schema, <op-name>-REQUEST, includes the Θ schema in its signature. The predicate expresses the constraints on the personae dramatis which one expects of a reasonable request simply in order to conform with the structure of the machine. Thus, if the signature includes a gamma entity and a join entity one expects the gamma object to have entity type Gamma, the join to have type Join and the gamma object to be in the g of the join entity, etc. These are important constraints and give the operation meaning in the context of the machines structure but they are not the obviously important overt security checks.

The third schema, GOOD-<op name>, includes <op-name>-REQUEST and its predicate includes the important checks and the main effects of the transition. This schema is thus the main essence of the transition. On first reading of a transition one is therefore mainly interested in the signature of Θ<*op-name*> and the predicate of GOOD_<op-name>. These alone are sufficient for gaining the essence of the command.

basic_types :Module	labels :Module	basic_state :Module

The Initiate Login Command

A login is instigated by a trusted path, port entity which is currently not logged in. This status is distinguished by whether or not a user entity is currently pointing at the port in g. Login provides the identity of the person logging in and the session clearance which they are claiming.

All that happens in this transition is that the requested values are lodged with the garbage collector together with sufficient data changes to allow the garbage collector to subsequently use the correct port, etc. The instigator is told that the request has been initiated.

ΦINITIATE_LOGIN

$\Delta STATE$
name : IDENTITY
session_labels : SECURITY_LABELS
port_label : E

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$

INITIATE_LOGIN_REQUEST

ΦINITIATE_LOGIN

$dom(g \rhd \{instigator\}) = \{port\_label, GC\}$
$entitytype(instigator) = Trusted\_Path$
$entitytype(port\_label) = Label\_Object$

GOOD_INITIATE_LOGIN

INITIATE_LOGIN_REQUEST
INITIATE_LOGIN_OK : DATUM

$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{INITIATE\_LOGIN\_OK\}$
$inner\_labels' = inner\_labels \oplus \{ GC \mapsto session\_labels \}$
$inner\_role' = inner\_role \oplus \{ GC \mapsto name \}$

A bad login can occur if for example a user attempts to login while already logged in.

$BAD\_INITIATE\_LOGIN \triangleq \Phi STATE_{[INITIATE\_LOGIN\_NOT\_OK/RESULT]}$

Overall initiate login is defined thus

$INITIATE\_LOGIN \triangleq BAD\_INITIATE\_LOGIN \oplus GOOD\_INITIATE\_LOGIN$

initiate_login keeps INITIATE_LOGIN

basic_types :Module	labels :Module	basic_state :Module

The GC Execution of Login

This is the transition which actually creates a trusted command line interpreter User process. This will be able to access in its g the trusted path port and the home directory. From its home directory it can access root and hence, subject to security restrictions, the rest of the system.

The name and session level for the logged in User are obtained from the inner labels of the garbage collector where they have been lodged by the initiate login command. The manner in which the garbage collector obtains the port, port label, home directory, the right subset of user authorisations etc, are all underspecified.

ΦLOGIN

$\Delta STATE$
$name : IDENTITY$
$session\_labels : SECURITY\_LABELS$
$home : E$
$UserApprovals : \mathbb{P}\, E$
$user\_shell : E$
$port, port\_label : E$

$\{user\_shell\} \lhd g' = \{user\_shell\} \lhd g$
$outer\_labels' = outer\_labels \oplus \{user\_shell \mapsto session\_labels\}$
$entitytype' = entitytype \oplus \{user\_shell \mapsto User\}$
$role' = role \oplus \{user\_shell \mapsto name\}$
$\{user\_shell, instigator, port\} \lhd data' = \{user\_shell, instigator, port\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{user\_shell \mapsto session\_labels\}$
$inner\_role' = inner\_role \oplus \{user\_shell \mapsto name\}$

LOGIN_REQUEST

ΦLOGIN

$dom(g \rhd \{user\_shell\}) = \{GC\}$
$instigator = GC$
$dom(g \rhd \{port\}) = \{port\_label, GC\}$
$entitytype(port) = Trusted\_Path$
$entitytype(port\_label) = Label\_Object$
$entitytype(home) = Alpha$

User approvals are certificate objects lodged in root (indirectly via their label objects). User approvals point to a home directory which they are approving. The role of the certificate indicates the user so approving. The inner role and labels of the certificate indicate the role and clearance which that user approves of. The role and outer labels of the home directory indicate the role and clearance that the person who originally registered on the system wished to use. Login therefore requires that for every name in the conflict list of the clearance the user requested, a concurring approval from a user of that name exists.

Login allows logins below a users clearance. The User shell is then plumbed in and the port data updated with a login banner, etc. Note that a user shell points to itself when logged in. This is the indication to the garbage collector that the shell is active. The machine is such that nothing else (except the garbage collector) ever points at a user shell.

GOOD_LOGIN

LOGIN_REQUEST
LOGIN_OK : DATUM

$inner\_role(instigator) = name$
$inner\_labels(instigator) = session\_labels$
$UserApprovals \subseteq g(g(\{RT\})) \cap dom(entitytype \rhd \{Certificate\})$
$\{home\} = g(g(UserApprovals))$
$inner\_labels(UserApprovals) = \{outer\_labels(home)\}$
$inner\_role(UserApprovals) = \{role(home)\}$
$(outer\_labels(home)).conflict \subseteq role(UserApprovals)$
$outer\_labels(home)\ dominates\ session\_labels$
$name = role(home)$
$g'(\{user\_shell\}) = \{RTL, port, user\_shell, GC\} \cup g(UserApprovals)$
$data'(\{instigator\}) \subseteq data(\{instigator\}) \cup \{LOGIN\_OK\}$
$data'(\{port\}) \subseteq data(\{port\}) \cup \{LOGIN\_OK\}$
$data'(\{user\_shell\}) = \emptyset$

$BAD\_LOGIN \triangleq \Phi STATE_{[LOGIN\_NOT\_OK/RESULT]}$

$LOGIN \triangleq BAD\_LOGIN \oplus GOOD\_LOGIN$

login keeps LOGIN

basic_types :Module | basic_state :Module

The Logout Command

As one might expect therefore the reflexive link indication to the Garbage Collector discussed in the description of login leads to a particularly simple specification of logout. This simply removes the reflexive link in g at which point the shell becomes inactive and will eventually be removed by the garbage collector. However, this means that between logout and the garbage collector erasing the g of the user shell the port will be unavailable for further logins/registrations.

Logout also clears the data of the port object!

LOGOUT

$\Delta STATE$

$entitytype(instigator) = User$

$g' = g \setminus \{instigator \mapsto instigator\}$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$data' = (g(\{instigator\}) \cap dom(entitytype \rhd \{Trusted\_Path\})) \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

logout keeps LOGOUT

basic_types :Module | basic_state :Module

The Garbage Collector Command

To clarify the point that the inner shell is not immediately available after logoutwe can look at the specification of the garbage collector. This essentially does nothing but go through g removing entities which are not pointed to by anyone other than the garbage collector. Each pass therefore potentially generates more garbage and it is assumed that the garbage collection transition is invoked sufficiently frequently to keep garbage fully collected and available for allocation.

ΦGARBAGE_COLLECT
ΔSTATE

$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{GC\} \lhd data' = \{GC\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

GARBAGE_COLLECT_REQUEST
ΦGARBAGE_COLLECT

$instigator = GC$

GARBAGE_COLLECT
GARBAGE_COLLECT_REQUEST

$g' = garbage \lhd g$
where
$garbage == dom(\{GC\} \lhd g) \setminus rng(\{GC\} \lhd g)$

garbage_collector keeps GARBAGE_COLLECT

User Registration

So, we have seen how the system owner can login and logout using the initial structure of certificates and home directories. It may not seem immediately clear why such an elaborate mechanism is required and how new users are registered to create further such structures.

The SMITE policy is based on the notions of segregation of duties which further requires notions such as non-repudiation, assured auditing, etc, if normal social and judicial pressures are to be effective in providing motivation, deterrence, etc. Thus, in such a regime if the audit trail says I did it then a court of law will convict me. Hence, under such a system, the normal methods of user administration, where a super user creates a pseudo person loosely connected to the real person by a password login system, are no longer adequate.

This specification is assuming that IDENTITY values are non-repudiateable, non-forgeable tokens, such as in public key signature mechanisms, and is concerned in showing how the putative user and system administrator must co-operate in an intiallly mutually suspicious manner to create a pseudo user on the system. The specification deals only with initial mutual suspicion, at some point if you wish to use my system you have to sign and agree and trust some aspects of the way it operates. At least with this machine you cannot be put on it in the first place against your will.

The model to be captured is therefore that a person can, on walking up to one of the systems IO devices, offer a signed token of his identity and request use of the system under some role, (in this specification his identity but in general they need not be the same), at some clearance, under the responsibility of some group of users, which may or may not include himself but must always consist of at least two users. The system is not obliged to accept these grounds for use but it cannot change the terms without the cooperation of the user, thus it cannot grant access at a lower clearance or with a different set of responsible users. Similarly, the responsible users have to agree to be responsible for you in the same way that you would not want the system to suddenly make you responsible for something without your agreement.

This is achieved by user registration creating a home directory whose labels record the role and clearance and conflict list which the person requested. Pointing to this is a certificate, concurring with the labels of the home directory and owned by the person. Unlike the initial set up for the system owner this is insufficient for the person to log in because of the requirement that his conflict has at least two members. An existing user of the system must approve the request before he can log in. Initially, the first users on the system must nominate the system owner. It is envisaged therefore that the system owner, using physical access controls to the IO devices, initially enrolls a number of his responsible employees using mutual conflicts. These managers can subsequently manage the system and indeed the system owner could delete himself from the system. Using these simple mechanisms any separation of duties scheme can be built from the initial state.

basic_types :Module	labels :Module	basic_state :Module

The Initiate Registration Command

As with login this transition simply lodges with the garbage collector the parameters of the users request, a name and a clearance.

ΦINITIATE_USER_REGISTRATION

$\Delta STATE$
name : IDENTITY
clearance : SECURITY_LABELS
port_label : E

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto clearance\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto name\}$

INITIATE_USER_REGISTRATION_REQUEST
ΦINITIATE_USER_REGISTRATION

$entitytype(instigator) = Trusted\_Path$
$entitytype(port\_label) = Label\_Object$

The port must be idle and the requested conflict list must include some other user.

GOOD_INITIATE_USER_REGISTRATION
INITIATE_USER_REGISTRATION_REQUEST
INITIATE_USER_REGISTRATION_OK : DATUM

$dom(g \rhd \{instigator\}) = \{port\_label, GC\}$
$\#(clearance.conflict) > 1$
$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{INITIATE\_USER\_REGISTRATION\_OK\}$

$BAD\_INITIATE\_USER\_REGISTRATION \triangleq \Phi STATE_{[INITIATE\_USER\_REGISTRATION\_NOT\_OK/RESULT]}$

$INITIATE\_USER\_REGISTRATION \triangleq BAD\_INITIATE\_USER\_REGISTRATION \oplus GOOD\_INITIATE\_USER\_REGISTRATION$

initiate_user_registration keeps INITIATE_USER_REGISTRATION

basic_types :Module | labels :Module | basic_state :Module

The GC Execution of Registration

This is the transition which actually allocates a home directory, the initial users certificate and the two associated label objects. These are lodged in root.

ΦUSER_REGISTRATION

$\Delta STATE$
$home, home\_label : E$
$certificate, certificate\_label : E$

$\{RT, home, home\_label, certificate, certificate\_label\} \lhd g' = \{RT, home, home\_label, certificate, certificate\_label\} \lhd g$
$\{home, home\_label, certificate, certificate\_label\} \lhd outer\_labels' = \{home, home\_label, certificate, certificate\_label\} \lhd outer\_labels$
$entitytype' = entitytype \oplus \{home \mapsto Alpha, home\_label \mapsto Label\_Object, certificate\_label \mapsto Label\_Object, certificate \mapsto Certificate\}$
$\{home, home\_label, certificate, certificate\_label\} \lhd role' = \{home, home\_label, certificate, certificate\_label\} \lhd role$
$data' = \{home, home\_label, certificate\_label, certificate\} \lhd data$
$\{home, home\_label, certificate, certificate\_label\} \lhd inner\_labels' = \{home, home\_label, certificate, certificate\_label\} \lhd inner\_labels$
$\{home, home\_label, certificate, certificate\_label\} \lhd inner\_role' = \{home, home\_label, certificate, certificate\_label\} \lhd inner\_role$

USER_REGISTRATION_REQUEST
ΦUSER_REGISTRATION

$instigator = GC$
$dom(g \rhd \{home\_label, certificate\_label, certificate, home\}) = \{GC\}$

The significant fields are the role, the inner labels and the inner role of the certificate, and the outer labels and role of the home directory. All other fields are initialised to complete the sanitisation of the allocated entities.

GOOD_USER_REGISTRATION
USER_REGISTRATION_REQUEST
$USER\_REGISTRATION\_OK : DATUM$

$g'(\{RT\}) = g(\{RT\}) \cup \{certificate\_label\}$
$g'(\{certificate\_label\}) = \{certificate\}$
$g'(\{certificate\}) = \{home\_label\}$
$g'(\{home\_label\}) = \{home\}$
$g'(\{home\}) = \{RTL\}$
$outer\_labels'(\{home\_label, certificate, certificate\_label\}) = \{\bot\}$
$outer\_labels'(home) = inner\_labels(instigator)$
$role'(\{home, home\_label, certificate, certificate\_label\}) = \{inner\_role(instigator)\}$
$inner\_labels'(\{home, home\_label, certificate, certificate\_label\}) = \{inner\_labels(instigator)\}$
$inner\_role'(\{home, home\_label, certificate, certificate\_label\}) = \{inner\_role(instigator)\}$

$BAD\_USER\_REGISTRATION \triangleq \Phi STATE_{[USER\_REGISTRATION\_NOT\_OK/RESULT]}$

$USER\_REGISTRATION \triangleq BAD\_USER\_REGISTRATION \oplus GOOD\_USER\_REGISTRATION$

user_registration keeps USER_REGISTRATION

Reviewing and Approving Registrations

A new, putative user has thus registered a desire to use the system. How does an existing user review and approve the request?

First, he must gain access to the certificate. A user in his home directory has access to the label of root. To access root he must first "open" the label object. In root will be the label object of a users certificate. To open this he must first get the label object into his own g. This involves the "examine" command which enables a user to augment his g with elements from the g of an alpha object. He can then open the label of the certificate which puts the certificate in his g where it can now be reviewed.

Review copies the data and inner labels and role of a certificate into the user entity where we can assume it can be subsequently displayed on the trusted path. If the human user is satisified he can instruct the user command line entity to authorise the certificate. This involves the creation of a copy of the original certificate except that the outer role indicates the agreement of the existing user. To allow login to succeed this certificate (strictly its associated label) must be lodged in root. As with the other create commands approve certificate allows the parent node which will hold the label object to be specified.

In the following we will look at the open, examine, review certificate, and authorise certificate commands. The reading and writing of the trusted path by the user shell is elided from this description. These commands are covered later.

basic_types :Module | labels :Module | basic_state :Module

The Open Command

A "labelled object" is actually a pair of objects, a label object pointing in g at another object. Opening a "labelled object" is the act of copying the g reference to the object from the label object to the instigators g.

ΦUSER_OPEN

$\Delta STATE$
label_object : E
labelled_object : E

$\{instigator\} \lhd g' = \{instigator\} \lhd g$
outer_labels' = outer_labels
entitytype' = entitytype
role' = role
$\{instigator\} \lhd data' = \{instigator\} \lhd data$
inner_labels' = inner_labels
inner_role' = inner_role

USER_OPEN_REQUEST

ΦUSER_OPEN

---

entitytype(instigator) = User
entitytype(label_object) = Label_Object
$\{labelled\_object\} = g(\!| \{label\_object\} |\!)$

To open an object one must have the label object already in ones g. The labels of the user must also dominate the labels of the object to be placed in his g. This is checked not with the outer labels of the labelled object but with the inner labels of the label object. It is an invariant property of the machine that this equals the outer labels of the labelled object. This indirection is necessary for our proof approach. The instigators g may be augmented or may overwrite an existing g entry for the instigator.

GOOD_USER_OPEN

USER_OPEN_REQUEST
USER_OPEN_OK : DATUM

$label\_object \in g(\!| \{instigator\} |\!)$
outer_labels(instigator) dominates inner_labels(label_object)
$g'(\!|\{instigator\}|\!) \subseteq g(\!|\{instigator\}|\!) \cup \{labelled\_object\}$
$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{USER\_OPEN\_OK\}$

$BAD\_USER\_OPEN \triangleq \Phi STATE_{[USER\_OPEN\_NOTOK/RESULT]}$

$USER\_OPEN \triangleq BAD\_USER\_OPEN \oplus GOOD\_USER\_OPEN$

user_open keeps USER_OPEN

basic_types :Module | basic_state :Module

The Examine Command

Unlike a label object the labelled object may contain many entries in its g. The examine command allows the instigator to select from amongst these an entry with which to augment its own g. It is this command which allows the instigator to traverse the branching structure of the alpha hierarchy. There is no explicit security level check for examine because the open command has already checked the levels before placing the object in our g. Thus, it is a property of the machine that any alpha object in the g of a user is dominated by the user.

$\Phi USER\_EXAMINE\_ALPHA$

$\Delta STATE$

$alpha\_object : E$

$\{instigator\} \lhd g' = \{instigator\} \lhd g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

$USER\_EXAMINE\_ALPHA\_REQUEST$

$\Phi USER\_EXAMINE\_ALPHA$

$entitytype(instigator) = User$

$entitytype(alpha\_object) = Alpha$

$GOOD\_USER\_EXAMINE\_ALPHA$

$USER\_EXAMINE\_ALPHA\_REQUEST$

$USER\_EXAMINE\_ALPHA\_OK : DATUM$

$alpha\_object \in g(\!| \{instigator\} |\!)$

$g'(\!|\{instigator\}|\!) \subseteq g(\!|\{instigator, alpha\_object\}|\!)$

$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{USER\_EXAMINE\_ALPHA\_OK\}$

$BAD\_USER\_EXAMINE\_ALPHA \triangleq \Phi STATE_{[USER\_EXAMINE\_ALPHA\_NOTOK/RESULT]}$

$USER\_EXAMINE\_ALPHA \triangleq BAD\_USER\_EXAMINE\_ALPHA \oplus GOOD\_USER\_EXAMINE\_ALPHA$

$user\_examine\_alpha$ keeps $USER\_EXAMINE\_ALPHA$

basic_types :Module | basic_state :Module

The Review Certificate Command

This command enables a user to peruse the data value aspects of a certificate, namely, data, inner_labels and inner_role. This command is used for both user login authentication certificates and downgrade certificates which is where the data values are primarily of use.

ΦREVIEW_CERTIFICATE

$\Delta STATE$

$certificate : E$

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$\{instigator\} \lhd inner\_labels' = \{instigator\} \lhd inner\_labels$

$\{instigator\} \lhd inner\_role' = \{instigator\} \lhd inner\_role$

REVIEW_CERTIFICATE_REQUEST

ΦREVIEW_CERTIFICATE

$entitytype(instigator) = User$

$entitytype(certificate) = Certificate$

$certificate \in g(\{instigator\})$

Again no explicit security check is required because to be in the users g the certificate must have already been opened or created there.

GOOD_REVIEW_CERTIFICATE

REVIEW_CERTIFICATE_REQUEST

$REVIEW\_CERTIFICATE\_OK : DATUM$

$data'(\{instigator\}) \subseteq data(\{instigator, certificate\}) \cup \{REVIEW\_CERTIFICATE\_OK\}$

$inner\_labels'(instigator) = inner\_labels(certificate)$

$inner\_role'(instigator) = inner\_role(certificate)$

$BAD\_REVIEW\_CERTIFICATE \triangleq \Phi STATE_{[REVIEW\_CERTIFICATE\_NOT\_OK/RESULT]}$

$REVIEW\_CERTIFICATE \triangleq BAD\_REVIEW\_CERTIFICATE \oplus GOOD\_REVIEW\_CERTIFICATE$

review_certificate keeps REVIEW_CERTIFICATE

basic_types :Module	labels :Module	basic_state :Module

The Initiate Certificate Approval Command

Approval of a certificate simply involves creating a copy of it with your own role instead of that of the certificate you are approving. As this involves allocation of a new certificate entity this command is again modelled as a two part command involving the garbage collector.

ΦINIT_APPROVE_CERTIFICATE

$\Delta STATE$
certificate, parent : *E*
certificate_label : *SECURITY_LABELS*

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$

$\{GC, instigator\} \lhd data' = \{GC, instigator\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto certificate\_label\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto role(instigator)\}$

The information lodged is the role of the instigator and the label for the new certificate. The specification of the GC data changing is to allow the necessary implementation details, which record the parent alpha object where the certificate is to be lodged, etc, without obscuring the specification with details.

INIT_APPROVE_CERTIFICATE_REQUEST
ΦINIT_APPROVE_CERTIFICATE

$entitytype(instigator) = User$
$entitytype(parent) = Alpha$
$entitytype(certificate) = Certificate$
$certificate \in g(\{instigator\})$
$parent \in g(\{instigator\})$

GOOD_INIT_APPROVE_CERTIFICATE
INIT_APPROVE_CERTIFICATE_REQUEST
INIT_APPROVE_CERTIFICATE_OK : *DATUM*

$data'(\{instigator\}) \subseteq data(\{instigator\}) \cup \{INIT\_APPROVE\_CERTIFICATE\_OK\}$
certificate_label dominates outer_labels(certificate)

The label requested for the new certificate must dominate the existing certificate in order to protect the information contained to the same level as the object from which it was obtained. We don't want premature downgrading!

$BAD\_INIT\_APPROVE\_CERTIFICATE \triangleq \Phi STATE_{[INIT\_APPROVE\_CERTIFICATE\_NOT\_OK/RESULT]}$

$INIT\_APPROVE\_CERTIFICATE \triangleq BAD\_INIT\_APPROVE\_CERTIFICATE \oplus GOOD\_INIT\_APPROVE\_CERTIFICATE$

init_approve_certificate keeps INIT_APPROVE_CERTIFICATE

basic_types :Module	labels :Module	basic_state :Module

The Approve Certificate Command

This is the transition which actually creates the new certificate. Both a certificate and a label object for it are created hence the apparent complexity, because the two objects must be stitched together in g starting from the parent alpha object and ending with the target alpha object's label object, and all of their fields must be initialised appropriately.

ΦAPPROVE_CERTIFICATE

$\Delta STATE$
certificate, parent : E
new_certificate, label_object : E

$\{parent, new\_certificate, label\_object\} \lhd g' = \{parent, new\_certificate, label\_object\} \lhd g$
$\{new\_certificate, label\_object\} \lhd outer\_labels =$
$\{new\_certificate, label\_object\} \lhd outer\_labels$
$entitytype' = entitytype \oplus \{label\_object \mapsto Label\_Object, new\_certificate \mapsto Certificate\}$
$\{new\_certificate, label\_object\} \lhd role' = \{new\_certificate, label\_object\} \lhd role$
$\{instigator, new\_certificate, label\_object\} \lhd data' =$
$\{instigator, new\_certificate, label\_object\} \lhd data$
$\{new\_certificate, label\_object\} \lhd inner\_labels =$
$\{new\_certificate, label\_object\} \lhd inner\_labels$
$\{new\_certificate, label\_object\} \lhd inner\_role' = \{new\_certificate, label\_object\} \lhd inner\_role$

APPROVE_CERTIFICATE_REQUEST
ΦAPPROVE_CERTIFICATE

$instigator = GC$
$entitytype(parent) = Alpha$
$entitytype(certificate) = Certificate$
$dom(g \rhd \{label\_object, new\_certificate\}) = \{GC\}$

The parameters which we have obtained from the initiating command by unspecified means involving the garbage collectors data fields are the certificate to be approved and the parent alpha object where the new certificate is to be lodged.

GOOD_APPROVE_CERTIFICATE
APPROVE_CERTIFICATE_REQUEST
APPROVE_CERTIFICATE_OK : *DATUM*

$g'(\!|\{parent\}|\!) \subseteq g(\!|\{parent\}|\!) \cup \{label\_object\}$
$g'(\!|\{label\_object\}|\!) = \{new\_certificate\}$
$g'(\!|\{new\_certificate\}|\!) = g(\!|\{certificate\}|\!)$
$outer\_labels(parent)$ *dominates* $outer\_labels'(label\_object)$
$outer\_labels'(new\_certificate) = inner\_labels(instigator)$
$role'(\!|\{label\_object, new\_certificate\}|\!) = \{inner\_role(instigator)\}$
$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{APPROVE\_CERTIFICATE\_OK\}$
$data'(\!|\{new\_certificate\}|\!) = data(\!|\{certificate\}|\!)$
$data'(\!|\{label\_object\}|\!) = \emptyset$
$inner\_labels'(new\_certificate) = inner\_labels(certificate)$
$inner\_labels'(label\_object) = outer\_labels'(new\_certificate)$
$inner\_role'(new\_certificate) = inner\_role(certificate)$
$inner\_role'(label\_object) = SysOwner$

So the parent is set to point to the new label object, which points at the new certificate which points at the same things as the existing certificate. The outer_labels of the new label object must be dominated by the labels of the parent in order to preserve the property of the machine that allows open/examine to traverse the alpha structure without signalling channels. The outer labels of the new certificate, and hence the inner labels of the label object, are set as requested in the initiating command which has been lodged in the inner labels of the garbage collector. Similarly, the role of the new certificate reflects the original instigator,

temporarily lodged in the garbage collector's inner role. The setting of all other fields is straighforward, either a copy from the original certificate or an initialisation to a don't care value which ensures sanitisation of the new objects.

$BAD\_APPROVE\_CERTIFICATE \triangleq \Phi STATE_{[APPROVE\_CERTIFICATE\_NOT\_OK/RESULT]}$

$APPROVE\_CERTIFICATE \triangleq BAD\_APPROVE\_CERTIFICATE \oplus GOOD\_APPROVE\_CERTIFICATE$

approve_certificate keeps APPROVE_CERTIFICATE

The Basic User Commands on the Alpha Hierarchy

Having seen how users are registered and logged in we can now explore some more of the commands available to them. We have seen how users can open label objects to obtain access to the alpha or certificate objects behind them, and how the g of alpha objects can be extracted to augment the users g, effectively a traversal of the alpha structure by successive open/examine pairs. The other commands available are the obvious read and write of alpha objects, and the creation and deletion of alpha objects together with their associated label objects. Users can also delete certificates.

basic_types :Module | basic_state :Module

The Read Alpha Command for a User

As before no explict security check is required other than that the alpha object is in the user's g because of the open check which preserves the required security property of the machine.

ΦUSER_READ_ALPHA

$\Delta STATE$

alpha_object ? : *ALPHA*

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

USER_READ_ALPHA_REQUEST

ΦUSER_READ_ALPHA

$entitytype(instigator) = User$

$entitytype(alpha\_object) = Alpha$

GOOD_USER_READ_ALPHA

USER_READ_ALPHA_REQUEST

USER_READ_ALPHA_OK : *DATUM*

$alpha\_object \in g(\{instigator\})$

$data'(\{instigator\}) \subseteq data(\{instigator, alpha\_object\}) \cup \{USER\_READ\_ALPHA\_OK\}$

$BAD\_USER\_READ\_ALPHA \triangleq \Phi STATE_{[USER\_READ\_ALPHA\_NOTOK/RESULT]}$

$USER\_READ\_ALPHA \triangleq BAD\_USER\_READ\_ALPHA \oplus GOOD\_USER\_READ\_ALPHA$

user_read_alpha keeps USER_READ_ALPHA

basic_types :Module | basic_state :Module | labels :Module

The Write Alpha Command for a User

While no security check for write downs is required here, because users are defined to be complex sources in the formal policy formulation of this abstract machine, we nonetheless impose such a check. This is because we envisage the use of complex sourceness as a solution to the unavoidable covert channels involved in the user creating and deleting shared alpha objects not as permission for a user to copy data wholesale. Thus for the overt data transfer we impose the typical no write down relative to the users clearance which is found in most extant models.

ΦUSER_WRITE_ALPHA

$\Delta STATE$
alpha_object : E

$g' = g$
outer_labels' = outer_labels
entitytype' = entitytype
role' = role
$\{instigator, alpha\_object\} \lhd data' = \{instigator, alpha\_object\} \lhd data$
inner_labels' = inner_labels
inner_role' = inner_role

USER_WRITE_ALPHA_REQUEST
ΦUSER_WRITE_ALPHA

---

entitytype(instigator) = User
entitytype(alpha_object) = Alpha

GOOD_USER_WRITE_ALPHA
USER_WRITE_ALPHA_REQUEST
USER_WRITE_ALPHA_OK : DATUM

$alpha\_object \in g(\!| \{instigator\} |\!)$
outer_labels(alpha_object) dominates outer_labels(instigator)
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{USER\_WRITE\_ALPHA\_OK\}$
$data'(\!|\{alpha\_object\}|\!) \subseteq data(\!|\{alpha\_object, instigator\}|\!)$

$BAD\_USER\_WRITE\_ALPHA \triangleq \Phi STATE_{[USER\_WRITE\_ALPHA\_NOTOK/RESULT]}$

$USER\_WRITE\_ALPHA \triangleq BAD\_USER\_WRITE\_ALPHA \oplus GOOD\_USER\_WRITE\_ALPHA$

user_write_alpha keeps USER_WRITE_ALPHA

basic_types :Module	labels :Module	basic_state :Module

The Initiate Create Alpha Command

Creation of an alpha entity requires allocation of a new alpha entity and its associated label object so once again this transition is specified in two parts involving the gabage collector. Again intermediate information is lodged explicitly in the inner controls or implicitly in the garbage collectors data structures.

The information lodged consists of the parent alpha object where the new object is to be inserted and the label of the new object.

Once again a no write down check is not formally required on this label because users are deemed complex sources and this time we wish to utilise this fact to allow the useful behaviour of a user.

$\Phi INIT\_USER\_CREATE\_ALPHA$

$\Delta STATE$
$parent : E$
$label : SECURITY\_LABELS$

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto label\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto role(instigator)\}$

$INIT\_USER\_CREATE\_ALPHA\_REQUEST$
$\Phi INIT\_USER\_CREATE\_ALPHA$

$entitytype(instigator) = User$
$entitytype(parent) = Alpha$
$parent \in g(\{instigator\})$

$GOOD\_INIT\_USER\_CREATE\_ALPHA$
$INIT\_USER\_CREATE\_ALPHA\_REQUEST$
$INIT\_USER\_CREATE\_ALPHA\_OK : DATUM$

$data'(\{instigator\}) = data(\{instigator\}) \cup \{INIT\_USER\_CREATE\_ALPHA\_OK\}$

$BAD\_INIT\_USER\_CREATE\_ALPHA \triangleq \Phi STATE_{\{INIT\_USER\_CREATE\_ALPHA\_NOTOK/RESULT\}}$

$INIT\_USER\_CREATE\_ALPHA \triangleq BAD\_INIT\_USER\_CREATE\_ALPHA \oplus GOOD\_INIT\_USER\_CREATE\_ALPHA$

init_user_create_alpha keeps INIT_USER_CREATE_ALPHA

basic_types :Module | labels :Module | basic_state :Module

The GC Create Alpha Command

As before the means by which the parent alpha object is conveyed between the intiating command and this command is underspecified. All other information is lodged in the inner controls of the garbage collector.

ΔUSER_CREATE_ALPHA

$\Delta STATE$
$label\_object : E$
$alpha\_object : E$
$parent : E$

$\{parent, label\_object, alpha\_object\} \lhd g' = \{parent, label\_object, alpha\_object\} \lhd g$
$\{alpha\_object, label\_object\} \lhd outer\_labels' = \{alpha\_object, label\_object\} \lhd outer\_labels$
$entitytype' = entitytype \oplus \{label\_object \mapsto Label\_Object, alpha\_object \mapsto Alpha\}$
$role' = role \oplus \{label\_object \mapsto inner\_role(instigator),$
$\quad alpha\_object \mapsto inner\_role(instigator)\}$
$\{instigator, label\_object, alpha\_object\} \lhd data' =$
$\{instigator, label\_object, alpha\_object\} \lhd data$
$\{label\_object, alpha\_object\} \lhd inner\_labels' =$
$\{label\_object, alpha\_object\} \lhd inner\_labels$
$inner\_role' = inner\_role \oplus \{label\_object \mapsto inner\_role(instigator),$
$\quad alpha\_object \mapsto inner\_role(instigator)\}$

USER_CREATE_ALPHA_REQUEST

$\Phi USER\_CREATE\_ALPHA$

$dom(g \rhd \{label\_object\}) = \{GC\}$
$dom(g \rhd \{alpha\_object\}) = \{GC\}$
$instigator = GC$
$entitytype(parent) = Alpha$

GOOD_USER_CREATE_ALPHA

$USER\_CREATE\_ALPHA\_REQUEST$
$USER\_CREATE\_ALPHA\_OK : DATUM$

$g'(\{parent\}) = g(\{parent\}) \cup \{label\_object\}$
$g'(\{label\_object\}) = \{alpha\_object\}$
$g'(\{alpha\_object\}) = \emptyset$
$outer\_labels(parent)\ dominates\ outer\_labels'(label\_object)$
$outer\_labels'(alpha\_object) = inner\_labels(instigator)$
$data'(\{instigator\}) = data(\{instigator\}) \cup \{USER\_CREATE\_ALPHA\_OK\}$
$data'(\{label\_object\}) = \emptyset$
$data'(\{alpha\_object\}) = \emptyset$
$inner\_labels'(label\_object) = inner\_labels(instigator)$
$inner\_labels'(alpha\_object) = inner\_labels(instigator)$

The new alpha object label object pair are stiched into g starting at the parent, which points at the new label object, which in turn points at the new alpha object, which has an empty g.

The requested outer labels of the new alpha object are lodged in the garbage collectors inner labels.

The outer label of the label object is not specified other than it must be dominated by the parent's label, in order to preserve the security property of the alpha storage structure. In reality the user would wish to specify this value, but, in the same way that a user is a complex source and he can therefore request any label consistent with his clearance, as he has in this case for the alpha object itself, there is no real check which we need to enforce on this value. This is not therefore a security oversight. It must however be assigned a user selected value if there is not to be a storage channel through an unitialised variable when the new entity is allocated.

$BAD\_USER\_CREATE\_ALPHA \triangleq \Phi STATE_{[USER\_CREATE\_ALPHA\_NOTOK/RESULT]}$

USER_CREATE_ALPHA ≙ BAD_USER_CREATE_ALPHA ⊕ GOOD_USER_CREATE_ALPHA

user_create_alpha keeps USER_CREATE_ALPHA

basic_types :Module | basic_state :Module

The Delete Alpha Command for Users

This command simply removes a label object from an alpha object's g. This only leads to actual deallocation of the entities when the garbage collector detects no references to them. Because a user is a complex source there is not explicit security check for writing down when the parent object's g is modified.

ΦUSER_DELETE_ALPHA

$\Delta STATE$
$parent : E$
$old\_label\_object : E$
$old\_alpha\_object : E$

$\{parent\} \lhd g' = \{parent\} \lhd g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator\} \lhd data' = \{instigator\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

USER_DELETE_ALPHA_REQUEST

$\Phi USER\_DELETE\_ALPHA$

$entitytype(instigator) = User$
$entitytype(parent) = Alpha$
$entitytype(old\_label\_object) = Label\_Object$
$entitytype(old\_alpha\_object) = Alpha$
$old\_label\_object \in g(\!|\{parent\}|\!)$
$old\_alpha\_object \in g(\!|\{old\_label\_object\}|\!)$

GOOD_USER_DELETE_ALPHA

$USER\_DELETE\_ALPHA\_REQUEST$
$USER\_DELETE\_ALPHA\_OK : DATUM$

$parent \in g(\!|\{instigator\}|\!)$
$g'(\!|\{parent\}|\!) = g(\!|\{parent\}|\!) \setminus \{old\_label\_object\}$
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{USER\_DELETE\_ALPHA\_OK\}$

$BAD\_USER\_DELETE\_ALPHA \triangleq \Phi STATE_{[USER\_DELETE\_ALPHA\_NOTOK/RESULT]}$

$USER\_DELETE\_ALPHA \triangleq BAD\_USER\_DELETE\_ALPHA \oplus GOOD\_USER\_DELETE\_ALPHA$

user_delete_alpha keeps USER_DELETE_ALPHA

basic_types :Module | basic_state :Module

The Delete Certificate Command

A certificate is a variant of an alpha object in many respects so this command is almost identical to delete alpha except that, for integrity purposes, it requires that a user can only delete his own certificates and that the certificates pointer to the target entity is also removed. Conditional on entitytype this command could be elided with delete alpha in any real implementation.

The entities of concern are the parent alpha object in which the certificate, or more precisely its label object is lodged, the certificate and its label object, and finally the alpha object which is the subject of the certificate.

ΦUSER_DELETE_CERTIFICATE

$\Delta STATE$
$parent : E$
$certificate : E$
$label\_object : E$
$target : E$

$\{parent, certificate\} \lhd g' = \{parent, certificate\} \lhd g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator\} \lhd data' = \{instigator\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

USER_DELETE_CERTIFICATE_REQUEST
ΦUSER_DELETE_CERTIFICATE

$entitytype(instigator) = User$
$entitytype(parent) = Alpha$
$entitytype(label\_object) = Label\_Object$
$entitytype(certificate) = Certificate$
$entitytype(target) = Label\_Object$
$label\_object \in g(\{parent\})$
$certificate \in g(\{label\_object\})$
$target \in g(\{certificate\})$

As always the parent must be in the g of the instigator and the certificate must be one of the instigator's, in which case the parent to label object and certificate to alpha object links are removed from the g relation.

GOOD_USER_DELETE_CERTIFICATE
USER_DELETE_CERTIFICATE_REQUEST
USER_DELETE_CERTIFICATE_OK : DATUM

$parent \in g(\{instigator\})$
$role(instigator) = role(certificate)$
$g' = g \setminus \{ parent \mapsto label\_object, certificate \mapsto target \}$
$data'(\{instigator\}) = data(\{instigator\}) \cup \{USER\_DELETE\_CERTIFICATE\_OK\}$

$BAD\_USER\_DELETE\_CERTIFICATE \triangleq \Phi STATE_{[USER\_DELETE\_CERTIFICATE\_NOT\_OK/RESULT]}$

$USER\_DELETE\_CERTIFICATE \triangleq BAD\_USER\_DELETE\_CERTIFICATE \oplus GOOD\_USER\_DELETE\_CERTIFICATE$

user_delete_certificate keeps USER_DELETE_CERTIFICATE

The Use of Untrusted Software by Users

As described above the user processes, by virtue of being Trojan Horse free processes working under the direction of humans on a trusted path interface, are given great leeway in the actions which they can undertake, the creation and deletion of alpha entities for example, which in strict information flow terms violate a no flows down policy. This is allowed on the basis that human beings lack the patience, dedication, etc, to utilise complex signalling paths through a constrained interface. We do not assume however that, given the ability to copy a file wholesale, a subverted user would refrain from such action, even in the face of detailed auditing. Hence the write command enforces the usual no write down condition relative to the user's clearance.

As we have argued, such a constraint, which is the norm in extant policy model approaches, prevents users carrying out their assigned tasks without crippling overclassification of their results. Also the application specific manipulations modelled by the notions of simple read write are in practice more complex, spreadsheets, databases, etc. Given the complexity and application specific nature of these transitions our approach would be intractable if it consisted only of trusted users using trusted software to manipulate data.

Hence, we introduce the notion of untrusted software processes. These entities are assumed to be protected by floating labels and the no write down condition for these entities is relative to this floating label rather than the inherited clearance of the user. This allows untrusted software to get on with the job, without overclassifying results, assuming there is no trojan horse intervention.

To be successful this approach must assume that untrusted software trojan horses will attempt to signal, not only through shared data between users but also between gamma processes within the domain of a single user in order to circumvent the efficacy of the high water marks.

For these reasons, while the gamma entities are given the same functionality as the user, in terms of also being able to create and delete objects, for these entities stringent security controls are in place to ensure no flows down through signalling channels. In practice this means that such entities can only execute such functionality within areas of the machine where such signalling is hidden.

The model of operation envisaged which maximises the synergy between human users exercising discretion and untrusted software being constrained is that users create the environment of directories and files which create shared information paths and launch untrusted applications with highly constrained, least privilege access to these structures to achieve actual data manipulation.

We shall first consider the creation and deletion of gamma entities by user entities, then examine the variants of the commands available to gamma entities, and then look at the interaction between user and gamma entities.

These latter commands are modelled as simply the reading and writing of gamma entities by user entities. These are intended to cover the minor interactions of passing parameters on initiation, receiving results on completion and mediating interaction with the human user using the trusted path. They are not intended to imply a wholesale copying of data between user and gamma entities.

If this were allowed we would once again be in the scenario of a subverted user being able to copy data wholesale in a few simple steps. A gamma machine reads the alpha secrets and has its high water mark raised. The user copies the secrets to itself and then writes to another gamma entity with a low high water mark. The user interaction does not automatically raise a high water mark thus the second gamma could now write to a low alpha object achieving the subverted user's aim of releasing data.

It is not possible to convey this semantic distinction without modelling explicitly the nature of the individual user gamma entity interactions. In this general specification of the approach this is not feasible so this distinction is left implied. For the formal approach of an operational system these distinctions would need to be spelt out.

basic_types :Module | labels :Module | basic_state :Module

The Initiate Create Gamma Command

This again is a two part transition involving the gabage collector. The information lodged in this case is simply the role and labels of the user. The data of the garbage collector is also underspecified to imply the storage of the other necessary data to guide the command, primarily the required g of the new gamma entity.

ΦINIT_USER_CREATE_GAMMA
ΔSTATE

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto outer\_labels(instigator)\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto role(instigator)\}$

INIT_USER_CREATE_GAMMA_REQUEST
ΦINIT_USER_CREATE_GAMMA

$entitytype(instigator) = User$

GOOD_INIT_USER_CREATE_GAMMA
INIT_USER_CREATE_GAMMA_REQUEST
INIT_USER_CREATE_GAMMA_OK : DATUM

$data'(\{instigator\}) = data(\{instigator\}) \cup \{INIT\_USER\_CREATE\_GAMMA\_OK\}$

$BAD\_INIT\_USER\_CREATE\_GAMMA \triangleq \Phi STATE_{(INIT\_USER\_CREATE\_GAMMA\_NOTOK/RESULT)}$

$INIT\_USER\_CREATE\_GAMMA \triangleq BAD\_INIT\_USER\_CREATE\_GAMMA \oplus GOOD\_INIT\_USER\_CREATE\_GAMMA$

init_user_create_gamma keeps INIT_USER_CREATE_GAMMA

basic_types :Module	labels :Module	basic_state :Module

The Create Gamma Command

Because of the possibility of gamma entities being joined by a user a join entity recording this status must be allocated with the gamma entity.

The inner labels of the join entity is used to store the users clearance as an upper bound on the level to which the labels of the gamma machine are allowed to float. The outer labels of both the gamma and join entities are set to some level dominated by the users clearance, usually bottom, but implementations where the user sets some initial value are not ruled out. If the gamma machines g is not empty then the labels of the gamma and join entities must dominate all entities in the gamma object's g.

The implementation details of how these variations are controlled is implicitly covered by the data structures of the garbage collector which are deliberately underspecified for this purpose.

ΦUSER_CREATE_GAMMA

$\Delta STATE$
$join\_object : E$
$gamma\_object : E$
$user : E$

$\{user, join\_object, gamma\_object\} \lhd g' = \{user, join\_object, gamma\_object\} \lhd g$
$\{gamma\_object, join\_object\} \lhd outer\_labels' = \{gamma\_object, join\_object\} \lhd outer\_labels$
$entitytype' = entitytype \oplus \{ join\_object \mapsto Join, gamma\_object \mapsto Gamma \}$
$role' = role \oplus \{join\_object \mapsto inner\_role(instigator),$
$\quad gamma\_object \mapsto inner\_role(instigator) \}$
$\{gamma\_object, join\_object, instigator\} \lhd data' =$
$\{gamma\_object, join\_object, instigator\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{join\_object \mapsto inner\_labels(instigator),$
$\quad gamma\_object \mapsto inner\_labels(instigator)\}$

$inner\_role' = inner\_role \oplus \{join\_object \mapsto inner\_role(instigator),$
$\quad gamma\_object \mapsto inner\_role(instigator)\}$

USER_CREATE_GAMMA_REQUEST
ΦUSER_CREATE_GAMMA

$dom(g \rhd \{join\_object\}) = \{GC\}$
$dom(g \rhd \{gamma\_object\}) = \{GC\}$
$instigator = GC$
$entitytype(user) = User$

GOOD_USER_CREATE_GAMMA
USER_CREATE_GAMMA_REQUEST
USER_CREATE_GAMMA_OK : DATUM

$g'(\{user\}) = g(\{user\}) \cup \{join\_object, gamma\_object\}$
$g'(\{join\_object\}) = \{gamma\_object\}$
$g'(\{gamma\_object\}) \subseteq (g(\{user\}) \cap dom(entitytype' \rhd \{Alpha\_Label\_Object\}))$
$inner\_labels(instigator)$ *dominates* $outer\_labels'(gamma\_object)$
$outer\_labels'(join\_object) = outer\_labels'(gamma\_object)$
$inner\_labels'(join\_object) = inner\_labels(instigator)$
$\forall e : g'(\{gamma\_object\}) \bullet outer\_labels'(join\_object)$ *dominates* $outer\_labels(e)$
$data'(\{gamma\_object, join\_object\}) = \emptyset$

$BAD\_USER\_CREATE\_GAMMA \triangleq \Phi STATE_{[USER\_CREATE\_GAMMA\_NOTOK/RESULT]}$

$USER\_CREATE\_GAMMA \triangleq BAD\_USER\_CREATE\_GAMMA \oplus GOOD\_USER\_CREATE\_GAMMA$

user_create_gamma keeps USER_CREATE_GAMMA

basic_types :Module | basic_state :Module

The Delete Gamma Command

Because of the presence of the user join functionality we cannot permit the deletion of individual gamma entities but only joined groups. In practice therefore the command is actually to delete all references to the join entity. The garbage collector subsequently removes all unreferenced joined entities with no danger of unmodelled dangling references between joined objects which would otherwise exist.

ΦUSER_DELETE_GAMMA

$\Delta STATE$

$join\_object : E$

$dom(g \rhd \{join\_object\}) \lhd g' = dom(g \rhd \{join\_object\}) \lhd g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

USER_DELETE_GAMMA_REQUEST
ΦUSER_DELETE_GAMMA

$entitytype(instigator) = User$

$entitytype(join\_object) = Join$

GOOD_USER_DELETE_GAMMA
USER_DELETE_GAMMA_REQUEST
USER_DELETE_GAMMA_OK : DATUM

$join\_object \in g(\!|\{instigator\}|\!)$

$g' = g \rhd \{ join\_object \}$

$data' = data \cup \{ instigator \mapsto USER\_DELETE\_GAMMA\_OK \}$

$BAD\_USER\_DELETE\_GAMMA \triangleq \Phi STATE_{[USER\_DELETE\_GAMMA\_NO\_OK!RESULT]}$

$USER\_DELETE\_GAMMA \triangleq BAD\_USER\_DELETE\_GAMMA \oplus GOOD\_USER\_DELETE\_GAMMA$

user_delete_gamma keeps USER_DELETE_GAMMA

basic_types :Module | labels :Module | basic_state :Module

The Manipulation of Alpha Entities by Gamma Entities

The gamma machines can do basically what the user can do in terms of creating/deleting, reading/writing, opening/examining alpha objects. The security constraints are a little tighter and involve write down checks using the gamma machines floating label but the range of functionality is the same. The only things gamma machines absolutely cannot do are the certificate based operations of segregation of duties and the creation/deletion of gamma machines.

The Initiate Create Alpha for Gamma Entities

This is essentially as for user create alpha except that the label of the parent must dominate the floating label of the gamma entity which reflects the worst case sensitivity of the information they might be seeking to leak in modifying the parents g. This is the write down security check applied to untrusted code and from which the complex (trusted) users are exempt. As this check is carried out in the initiating command the actual creation of the alpha object is identical with user create alpha.

ΦINIT_GAMMA_CREATE_ALPHA

$\Delta STATE$
$parent : E$
$label : SECURITY\_LABELS$

$g' = g$
$outer\_labels' = outer\_labels'$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto label\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto role(instigator)\}$

INIT_GAMMA_CREATE_ALPHA_REQUEST
ΦINIT_GAMMA_CREATE_ALPHA

$entitytype(instigator) = Gamma$
$entitytype(parent) = Alpha$

GOOD_INIT_GAMMA_CREATE_ALPHA
INIT_GAMMA_CREATE_ALPHA_REQUEST
INIT_GAMMA_CREATE_ALPHA_OK : DATUM

$parent \in g(\{instigator\})$
outer_labels(parent) dominates outer_labels(instigator)
$data'(\{instigator\}) \subseteq data(\{instigator\}) \cup \{INIT\_GAMMA\_CREATE\_ALPHA\_OK\}$

$BAD\_INIT\_GAMMA\_CREATE\_ALPHA \triangleq \Phi STATE_{[INIT\_GAMMA\_CREATE\_ALPHA\_NOTOK/RESULT]}$

$INIT\_GAMMA\_CREATE\_ALPHA \triangleq BAD\_INIT\_GAMMA\_CREATE\_ALPHA \oplus GOOD\_INIT\_GAMMA\_CREATE\_ALPHA$

init_gamma_create_alpha keeps INIT_GAMMA_CREATE_ALPHA

basic_types :Module	labels :Module	basic_state :Module

The Delete Alpha Command for Gamma Entities

Again the only difference from the user version of the command is the application of a no write down check using the gamma entity's floating label and the modified parent.

ΦGAMMA_DELETE_ALPHA

$\Delta STATE$
$parent : E$
$old\_label\_object : E$
$old\_alpha\_object : E$

$\{parent\} \lhd g' = \{parent\} \lhd g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator\} \lhd data' = \{instigator\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

GAMMA_DELETE_ALPHA_REQUEST
ΦGAMMA_DELETE_ALPHA

$entitytype(instigator) = Gamma$
$entitytype(parent) = Alpha$
$entitytype(old\_label\_object) = Label\_Object$
$entitytype(old\_alpha\_object) = Alpha$
$old\_label\_object \in g(\!|\{parent\}|\!)$
$old\_alpha\_object \in g(\!|\{old\_label\_object\}|\!)$

GOOD_GAMMA_DELETE_ALPHA
GAMMA_DELETE_ALPHA_REQUEST
GAMMA_DELETE_ALPHA_OK : DATUM

$parent \in g(\!|\{instigator\}|\!)$
$outer\_labels(parent)$ *dominates* $outer\_labels(instigator)$
$g'(\!|\{parent\}|\!) = g(\!|\{parent\}|\!) \setminus \{old\_label\_object\}$
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{GAMMA\_DELETE\_ALPHA\_OK\}$

$BAD\_GAMMA\_DELETE\_ALPHA \triangleq \Phi STATE_{[GAMMA\_DELETE\_ALPHA\_NOTOK/RESULT]}$

$GAMMA\_DELETE\_ALPHA \triangleq BAD\_GAMMA\_DELETE\_ALPHA \oplus GOOD\_GAMMA\_DELETE\_ALPHA$

gamma_delete_alpha keeps GAMMA_DELETE_ALPHA

basic_types :Module	labels :Module	basic_state :Module

The Open Alpha Command for Gamma Entities

The "clearance" of a gamma entity is given by the inner_labels of its join entity. If it dominates the inner_labels of the label object the label of the join group is floated to the lub of its old value and the labelled objects classification. A joined groups floating label is actually the outer labels of all of the joined entities and the joining entity which must all be floated in unison.

ΦGAMMA_OPEN_ALPHA

$\Delta STATE$
label_object : *E*
alpha_object : *E*
join_object : *E*

$\{instigator\} \lhd g' = \{instigator\} \lhd g$
$(\{join\_object\} \cup g(\{join\_object\})) \lhd outer\_labels' =$
$(\{join\_object\} \cup g(\{join\_object\})) \lhd outer\_labels$
entitytype' = *entitytype*
role' = *role*
$\{instigator\} \lhd data' = \{instigator\} \lhd data$
inner_labels' = *inner_labels*
inner_role' = *inner_role*

GAMMA_OPEN_ALPHA_REQUEST
ΦGAMMA_OPEN_ALPHA

---

entitytype(instigator) = *Gamma*
entitytype(label_object) = *Label_Object*
entitytype(alpha_object) = *Alpha*
entitytype(join_object) = *Join*
instigator $\in g(\{join\_object\})$
alpha_object $\in g(\{label\_object\})$

GOOD_GAMMA_OPEN_ALPHA
GAMMA_OPEN_ALPHA_REQUEST
GAMMA_OPEN_ALPHA_OK : *DATUM*

---

label_object $\in g(\{instigator\})$
inner_labels(join_object) dominates inner_labels(label_object)
outer_labels'$(\{join\_object\} \cup g(\{join\_object\})) =$ {*lub* {*inner_labels(label_object)*, *outer_labels(instigator)* } }
alpha_object $\in g'(\{instigator\})$
data'$(\{instigator\}) = data(\{instigator\}) \cup \{GAMMA\_OPEN\_ALPHA\_OK\}$

$BAD\_GAMMA\_OPEN\_ALPHA \triangleq \Phi STATE_{[GAMMA\_OPEN\_ALPHA\_NOTOK/RESULT]}$

$GAMMA\_OPEN\_ALPHA \triangleq BAD\_GAMMA\_OPEN\_ALPHA \oplus GOOD\_GAMMA\_OPEN\_ALPHA$

gamma_open_alpha keeps GAMMA_OPEN_ALPHA

basic_types :Module | basic_state :Module

The Read Alpha Command for Gamma Entities

Apart from the type constraints this is identical to user read alpha and could be specified and implemented as such if desired.

$\Phi GAMMA\_READ\_ALPHA$

$\Delta STATE$

$alpha\_object : E$

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

$GAMMA\_READ\_ALPHA\_REQUEST$

$\Phi GAMMA\_READ\_ALPHA$

$entitytype(instigator) = Gamma$

$entitytype(alpha\_object) = Alpha$

$GOOD\_GAMMA\_READ\_ALPHA$

$GAMMA\_READ\_ALPHA\_REQUEST$

$GAMMA\_READ\_ALPHA\_OK : DATUM$

$alpha\_object \in g(\{instigator\})$

$data'(\{instigator\}) \subseteq data(\{instigator, alpha\_object\}) \cup \{GAMMA\_READ\_ALPHA\_OK\}$

$BAD\_GAMMA\_READ\_ALPHA \triangleq \Phi STATE_{[GAMMA\_READ\_ALPHA\_NOTOK/RESULT]}$

$GAMMA\_READ\_ALPHA \triangleq BAD\_GAMMA\_READ\_ALPHA \oplus GOOD\_GAMMA\_READ\_ALPHA$

gamma_read_alpha keeps GAMMA_READ_ALPHA

basic_types :Module	labels :Module	basic_state :Module

The Write Alpha Command for Gamma Entities

This simply requires the no write down check that the modified alpha entity dominates the floating label of the gamma entity. In all other respects (apart from type constraints) it is identical to user write alpha.

ΦGAMMA_WRITE_ALPHA

$\Delta STATE$

$alpha\_object : E$

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator, alpha\_object\} \lhd data' = \{instigator, alpha\_object\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

GAMMA_WRITE_ALPHA_REQUEST

ΦGAMMA_WRITE_ALPHA

$entitytype(instigator) = Gamma$

$entitytype(alpha\_object) = Alpha$

GOOD_GAMMA_WRITE_ALPHA

GAMMA_WRITE_ALPHA_REQUEST

GAMMA_WRITE_ALPHA_OK : DATUM

$alpha\_object \in g(\!| \{instigator\} |\!)$

$outer\_labels(alpha\_object)$ *dominates* $outer\_labels(instigator)$

$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{GAMMA\_WRITE\_ALPHA\_OK\}$

$data'(\!|\{alpha\_object\}|\!) \subseteq data(\!|\{alpha\_object, instigator\}|\!)$

$BAD\_GAMMA\_WRITE\_ALPHA \triangleq \Phi STATE_{[GAMMA\_WRITE\_ALPHA\_NOTOK/RESULT]}$

$GAMMA\_WRITE\_ALPHA \triangleq BAD\_GAMMA\_WRITE\_ALPHA \oplus GOOD\_GAMMA\_WRITE\_ALPHA$

gamma_write_alpha keeps GAMMA_WRITE_ALPHA

basic_types :Module | basic_state :Module

The Examine Alpha Command for Gamma Entities

Apart from the type constraints this is identical to user examine alpha and it could be specified and implemented as such.

$$
\begin{array}{|l}
\hline
\Phi GAMMA\_EXAMINE\_ALPHA \\
\Delta STATE \\
alpha\_object : E \\
\hline
\{instigator\} \lhd g' = \{instigator\} \lhd g \\
outer\_labels' = outer\_labels \\
entitytype' = entitytype \\
role' = role \\
\{instigator\} \lhd data' = \{instigator\} \lhd data \\
inner\_labels' = inner\_labels \\
inner\_role' = inner\_role \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
GAMMA\_EXAMINE\_ALPHA\_REQUEST \\
\Phi GAMMA\_EXAMINE\_ALPHA \\
\hline
entitytype(instigator) = Gamma \\
entitytype(alpha\_object) = Alpha \\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
GOOD\_GAMMA\_EXAMINE\_ALPHA \\
GAMMA\_EXAMINE\_ALPHA\_REQUEST \\
GAMMA\_EXAMINE\_ALPHA\_OK : DATUM \\
\hline
alpha\_object \in g(\!| \{instigator\} |\!) \\
g'(\!|\{instigator\}|\!) \subseteq g(\!|\{instigator, alpha\_object\}|\!) \\
data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{GAMMA\_EXAMINE\_ALPHA\_OK\} \\
\hline
\end{array}
$$

$$BAD\_GAMMA\_EXAMINE\_ALPHA \triangleq \Phi STATE_{[GAMMA\_EXAMINE\_ALPHA\_NOTOK/RESULT]}$$

$$GAMMA\_EXAMINE\_ALPHA \triangleq BAD\_GAMMA\_EXAMINE\_ALPHA \oplus GOOD\_GAMMA\_EXAMINE\_ALPHA$$

gamma_examine_alpha keeps GAMMA_EXAMINE_ALPHA

basic_types :Module | basic_state :Module

The Interaction of User and Gamma Entities

Obviously, gamma objects are only of use if the user shell can communicate results and commands to and from the gamma shells. Thus we have the reading and writing of gamma objects by the user shells. Also the joining of untrusted gamma entities so that for all security purposes the gamma machines act as one entity is useful for modelling the parameter passing nature of untrusted software.

The Read Gamma Command

Once again we see that the only security constraint here is that the object be in the g of the instigator. So apart from the type constraints this is identical to user read alpha and could be so specified and implemented.

ΦUSER_READ_GAMMA

$\Delta STATE$

$gamma\_object : E$

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

USER_READ_GAMMA_REQUEST

ΦUSER_READ_GAMMA

$entitytype(instigator) = User$

$entitytype(gamma\_object) = Gamma$

GOOD_USER_READ_GAMMA

USER_READ_GAMMA_REQUEST

$USER\_READ\_GAMMA\_OK : DATUM$

$gamma\_object \in g(\{instigator\})$

$data'(\{instigator\}) \subseteq (data(\{instigator, gamma\_object\}) \cup \{USER\_READ\_GAMMA\_OK\})$

$BAD\_USER\_READ\_GAMMA \triangleq \Phi STATE_{[USER\_READ\_GAMMA\_NOTOK/RESULT]}$

$USER\_READ\_GAMMA \triangleq BAD\_USER\_READ\_GAMMA \oplus GOOD\_USER\_READ\_GAMMA$

user_read_gamma keeps USER_READ_GAMMA

basic_types :Module	labels :Module	basic_state :Module

The Write Gamma Command

This command allows a user to write data to a gamma machine and/or bump up the join group's floating label to some new label. This latter feature may be required when the human user decides to enter some classified data into a file, for example.

ΦUSER_WRITE_GAMMA

$\Delta STATE$
$gamma\_object : E$
$join\_object : E$
$new\_label : SECURITY\_LABELS$

$g' = g$
$(\{join\_object\} \cup g(\!|\{join\_object\}|\!)) \lhd outer\_labels' = (\{join\_object\} \cup g(\!|\{join\_object\}|\!)) \lhd outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, gamma\_object\} \lhd data' = \{instigator, gamma\_object\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

USER_WRITE_GAMMA_REQUEST
ΦUSER_WRITE_GAMMA

$entitytype(instigator) = User$
$entitytype(gamma\_object) = Gamma$
$entitytype(join\_object) = Join$
$gamma\_object \in g(\!|\{join\_object\}|\!)$

GOOD_USER_WRITE_GAMMA
USER_WRITE_GAMMA_REQUEST
USER_WRITE_GAMMA_OK : DATUM

$gamma\_object \in g(\!|\{instigator\}|\!)$
$outer\_labels(instigator)\ dominates\ new\_label$
$new\_label\ dominates\ outer\_labels(join\_object)$
$outer\_labels'(\!|\{join\_object\} \cup g(\!|\{join\_object\}|\!)|\!) = \{new\_label\}$
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{USER\_WRITE\_GAMMA\_OK\}$
$data'(\!|\{gamma\_object\}|\!) \subseteq data(\!|\{gamma\_object, instigator\}|\!)$

$BAD\_USER\_WRITE\_GAMMA \triangleq \Phi STATE_{[USER\_WRITE\_GAMMA\_NOTOK/RESULT]}$

$USER\_WRITE\_GAMMA \triangleq BAD\_USER\_WRITE\_GAMMA \oplus GOOD\_USER\_WRITE\_GAMMA$

user_write_gamma keeps USER_WRITE_GAMMA

basic_types :Module | labels :Module | basic_state :Module

The Join Gamma Entities Command

In a join group the outer labels of the group float together and represent the high water mark of the group.

The inner label of the join entity of a group is the clearance of the group.

A user can refer to individual gamma machines or a group using his g entry for the join entity. In security terms all the entities of a group are one but when two groups are joined the user does not wish to lose his references to the individual subgroups for functionality purposes. When two groups are joined by reference to their join entities the "high water marks" as reflected by those two entities are lub'ed to produce the new high water marks for the new group.

The manner in which this is specified below means that such recursive joins result in individual subgroups within the group having disparate clearances. This is not a security problem. The clearances of a group can only be below the single clearance of the user who created them and who is the only one who can join them. The notion of giving a group a clearance below the users actual clearance is a least privilege notion which the user is choosing to override when joining groups with disparate clearances. For security purposes all that matters is that the high water mark is correct.

ΦUSER_JOIN_GAMMA

$\Delta STATE$

$join1, join2 : E$

$\{join1, join2\} \lhd g' = \{join1, join2\} \lhd g$

$(\{join1\ join2\} \cup g(\{join1\ join2\})) \lhd outer\_labels' = (\{join1\ join2\} \cup g(\{join1\ join2\})) \lhd outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

USER_JOIN_GAMMA_REQUEST

ΦUSER_JOIN_GAMMA

$entitytype(instigator) = User$

$entitytype(join1) = Join$

$entitytype(join2) = Join$

GOOD_USER_JOIN_GAMMA

USER_JOIN_GAMMA_REQUEST

USER_JOIN_GAMMA_OK : DATUM

$join1 \in g(\{instigator\})$

$join2 \in g(\{instigator\})$

$g'(\{join1\}) = g(\{join1\ join2\}) \cup \{join2\}$

$g'(\{join2\}) = g(\{join1\ join2\}) \cup \{join1\}$

$data'(\{instigator\}) = data(\{instigator\}) \cup \{USER\_JOIN\_GAMMA\_OK\}$

$outer\_labels'(\{join1\ join2\} \cup g(\{join1\ join2\})) = \{lub\ \{outer\_labels(join1), outer\_labels(join2)\}\}$

$BAD\_USER\_JOIN\_GAMMA \triangleq \Phi STATE_{(USER\_JOIN\_GAMMA\_NOTOK/RESULT)}$

$USER\_JOIN\_GAMMA \triangleq BAD\_USER\_JOIN\_GAMMA \oplus GOOD\_USER\_JOIN\_GAMMA$

user_join_gamma keeps USER_JOIN_GAMMA

Input and Output

Input and Ouput by the machine from/to human users or other machines is modelled by the trusted path port entities. These are modelled as active and can instigate the input transition where their data, as far as the specification goes, spontaneously changes.

This underspecification allows modelling of the changes to a keyboard buffer, etc. The trusted command line interpreter entities, users, can read and write the data of the trusted path entities. It is assumed that the data of a trusted path entitiy is in some way being displayed to the humans so the writing of data to the trusted path serves for modelling output.

The specification is trying to capture the idea of the complex, trusted command line interpreter maintaining a windows interface on the input/ouput device represented by the trusted path entity. The trusted path is therefore multi-level and trusted and is simply an extension of the complex user entity which it is convenient to model separately. For these reasons this specification does not make any attempt to use the controls of the trusted path entities to control any flows to/from the user entity. It could be extended to have devices which restrict the level at which users login etc and single level devices could be modelled. If we modelled a trusted path entity per window for example each window could be treated as a simple source with a high water mark, etc. These extensions simply require more commands for spawning windows, etc, under the control of the user command line interpreter entity.

basic_types :Module | basic_state :Module

The Input of Data Command

As stated this command simply wishes to capture, by underspecification, the fact that the input devices data changes, spontaneously as far as the machine is concerned but due to keyboard or mouse activity in reality.

If specified as simply letting data change this specification would allow the trusted devices to read the data of any entity in the system. In order to stress that these implementations are not intended we parachute in a new type representing data in the outside world and a conversion function to encode it as machine data. In this way, because there is no conversion from data to outputs to inputs and thereby back to data the specification forces an interpretation of genuine input.

$[INPUT]$

$input : INPUT \rightarrowtail\!\!\!\!\!\rightarrow DATUM$

GET_INPUT

$\Delta STATE$

$in? : INPUT$

$entitytype(instigator) = Trusted\_Path$

$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{input(in?)\}$

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$\{instigator\} \lhd data' = \{instigator\} \lhd data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

get_input keeps GET_INPUT

basic_types :Module	labels :Module	basic_state :Module

The Output Command

Similarly for output we wish to show that potentially any attribute of the trusted path device can be encoded for display to an external observer. So we define an output datatype with conversion functions for the defined attributes.

$OUTPUT ::= data\_out \langle\!\langle \mathbb{P}\, DATUM \rangle\!\rangle \mid label\_out \langle\!\langle SECURITY\_LABELS \rangle\!\rangle \mid role\_out \langle\!\langle IDENTITY \rangle\!\rangle \mid type\_out \langle\!\langle ENTITY\_TYPE \rangle\!\rangle$

PUT_OUTPUT

$\Delta STATE$

$out! : OUTPUT$

$entitytype(instigator) = Trusted\_Path$

$g' = g$

$outer\_labels' = outer\_labels$

$entitytype' = entitytype$

$role' = role$

$data' = data$

$inner\_labels' = inner\_labels$

$inner\_role' = inner\_role$

$out! \in \{ data\_out(data(\!|\{instigator\}|\!)), type\_out(entitytype(instigator)), role\_out(role(instigator)), role\_out(inner\_role(instigator)), label\_out(outer\_labels(instigator)), label\_out(inner\_labels(instigator))\}$

put_output keeps PUT_OUTPUT

basic_types :Module | basic_state :Module

Reading the Input Device

In reality the interface between a command line interpreter process and its IO devices would be highly structured as in the X protocols or the MaCHO interface. For simplicity in this top-level exposition of the abstract machine we simply model this as reading and writing the IO device.

$\Phi USER\_READ\_PATH$

$\Delta STATE$
$path : E$

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator\} \lhd data' = \{instigator\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

$USER\_READ\_PATH\_REQUEST$
$\Phi USER\_READ\_PATH$

$entitytype(instigator) = User$
$entitytype(path) = Trusted\_Path$

$GOOD\_USER\_READ\_PATH$
$USER\_READ\_PATH\_REQUEST$
$USER\_READ\_PATH\_OK : DATUM$

$path \in g(\!| \{instigator\} |\!)$
$data'(\!| \{instigator\} |\!) \subseteq (data(\!| \{instigator, path\} |\!) \cup \{USER\_READ\_PATH\_OK\})$

$BAD\_USER\_READ\_PATH \triangleq \Phi STATE_{[USER\_READ\_PATH\_NOTOK/RESULT]}$

$USER\_READ\_PATH \triangleq BAD\_USER\_READ\_PATH \oplus GOOD\_USER\_READ\_PATH$

user_read_path keeps USER_READ_PATH

basic_types :Module | basic_state :Module

Writing the Output Device

As for read path, this specification that the user command line interpreter process can simply write to the output device is a simplifying abstraction of what would in reality be a complex functional interface.

ΦUSER_WRITE_PATH

$\Delta STATE$
$path : E$

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, path\} \lhd data' = \{instigator, path\} \lhd data$
$inner\_labels' = inner\_labels$
$inner\_role' = inner\_role$

USER_WRITE_PATH_REQUEST
ΦUSER_WRITE_PATH

$entitytype(instigator) = User$
$entitytype(path) = Trusted\_Path$

GOOD_USER_WRITE_PATH
USER_WRITE_PATH_REQUEST
$USER\_WRITE\_PATH\_OK : DATUM$

$path \in g(\!| \{instigator\} |\!)$
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{USER\_WRITE\_PATH\_OK\}$
$data'(\!|\{path\}|\!) \subseteq data(\!|\{path, instigator\}|\!)$

$BAD\_USER\_WRITE\_PATH \triangleq \Phi STATE_{[USER\_WRITE\_PATH\_NOTOK/RESULT]}$

$USER\_WRITE\_PATH \triangleq BAD\_USER\_WRITE\_PATH \oplus GOOD\_USER\_WRITE\_PATH$

user_write_path keeps USER_WRITE_PATH

The Downgrading of Alpha Data

Last but not least we come to the downgrade commands. This is a separation of duty transition which, as we have seen in user registration, is modelled by the use of certificates.

Only the data of alpha objects can be downgraded and this involves copying the data to a new alpha entity in the alpha structure.

We do not allow downgrading by simply modifying the labels on an alpha entity because this alters the ability to traverse the tree structure as well as see the data. The information flow impact of this in terms of the proof analysis technique of "views" is intractable and the covert channels available through ordering make this unacceptable.

By using a copy to a new entity mechanism we introduce a functionality problem in that users do not wish to keep updating references to entities simply because they have been downgraded. This is a higher level naming resolution mechanism problem which we can model relatively easily because of the label object mechanisms already in place.

Downgrading is initiated by a user creating a certificate which points at the label object of the alpha object to be downgraded and which contains a snapshot of that objects data. This effects the freezing of data being downgraded.

Subsequent users can concurr with the downgrade by authorising existing certificates using review and authorise certificate, or by creating new certificates of their own. The danger with the latter approach is that when the actual commitment of the downgrade is attempted that the certificates will have captured different versions of the data because of changes to the document between certificate creations. If the certificates do not agree exactly on the data to be downgraded the downgrade will not occur.

The actual commit downgrade creates a new alpha object, copies the agreed certificate data to it and assigns the labels indicated by the certificates. It then redirects the old label object to this new alpha object and modifies the labels of the label object accordingly.

The approach to specifying certificates as holding copies of the data to be downgraded, protected by the labels of the certificate at the pre-downgrade level, means that the certificates must formally be regarded as complex sources, which is not really a surprise seeing as they are simply passive extensions of the human users expression of discretionary security decisions which are not controllable in terms of a normal lattice flow.

This problem could be ameliorated if desired by making certificates analogous to label objects rather than alpha objects. Thus a certificate would contain an indirection to an object containing the copy of the data to be downgraded. This would require however extending the details of the various data structures to hold the necessary labels, the new labels for the downgraded object, the labels of the copy of the putative downgraded data, etc. For the purposes of this exposition the general nature of using certificates is sufficient and such details have been avoided.

basic_types :Module	labels :Module	basic_state :Module

The Initiate Downgrade Request Command

Once again the creation of a certificate requires that this is modelled as a two part transition involving the garbage collector.

$\Phi INIT\_USER\_REQUEST\_DOWNGRADE$
$\Delta STATE$
$parent : E$
$target\_label : E$
$new\_label : SECURITY\_LABELS$

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto new\_label\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto role(instigator)\}$

$INIT\_USER\_REQUEST\_DOWNGRADE\_REQUEST$
$\Phi INIT\_USER\_REQUEST\_DOWNGRADE$

$entitytype(instigator) = User$
$entitytype(parent) = Alpha$
$entitytype(target\_label) = Label\_Object$

$GOOD\_INIT\_USER\_REQUEST\_DOWNGRADE$
$INIT\_USER\_REQUEST\_DOWNGRADE\_REQUEST$
$INIT\_USER\_REQUEST\_DOWNGRADE\_OK : DATUM$

$parent \in g(\!| \{instigator\} |\!)$
$target\_label \in g(\!| \{instigator\} |\!)$
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{INIT\_USER\_REQUEST\_DOWNGRADE\_OK\}$

$BAD\_INIT\_USER\_REQUEST\_DOWNGRADE \triangleq$
$\quad \Phi STATE_{[INIT\_USER\_REQUEST\_DOWNGRADE\_NOT\_OK/RESULT]}$

$INIT\_USER\_REQUEST\_DOWNGRADE \triangleq BAD\_INIT\_USER\_REQUEST\_DOWNGRADE$
$\quad \oplus GOOD\_INIT\_USER\_REQUEST\_DOWNGRADE$

init_user_request_downgrade keeps INIT_USER_REQUEST_DOWNGRADE

basic_types :Module | labels :Module | basic_state :Module

The GC Creation of the Downgrade Certificate

A certificate is treated as a variant of an alpha object and thus requires in the normal way that it possesses its own label object.

ΦUSER_REQUEST_DOWNGRADE

$\Delta STATE$
label_object : E
certificate : E
parent : E
target_label : E
target : E

$\{parent, label\_object, certificate\} \lhd g' = \{parent, label\_object, certificate\} \lhd g$
$\{certificate, label\_object\} \lhd outer\_labels' = \{certificate, label\_object\} \lhd outer\_labels$
$entitytype' = entitytype \oplus \{certificate \mapsto Certificate, label\_object \mapsto Label\_Object\}$
$\{certificate, label\_object\} \lhd role' = \{certificate, label\_object\} \lhd role$
$\{instigator, certificate, label\_object\} \lhd data' =$
$\{instigator, certificate, label\_object\} \lhd data$
$\{certificate, label\_object\} \lhd inner\_labels' = \{certificate, label\_object\} \lhd inner\_labels$
$\{certificate, label\_object\} \lhd inner\_role' = \{certificate, label\_object\} \lhd inner\_role$

USER_REQUEST_DOWNGRADE_REQUEST
ΦUSER_REQUEST_DOWNGRADE

$dom(g \rhd \{label\_object\}) = \{GC\}$
$dom(g \rhd \{certificate\}) = \{GC\}$
$instigator = GC$
$entitytype(parent) = Alpha$
$entitytype(target\_label) = Label\_Object$
$\{target\} = g(\!|\{target\_label\}|\!)$

GOOD_USER_REQUEST_DOWNGRADE

USER_REQUEST_DOWNGRADE_REQUEST
USER_REQUEST_DOWNGRADE_OK : DATUM

$outer\_labels(parent)\ dominates\ outer\_labels'(label\_object)$
$outer\_labels'(certificate) = inner\_labels(target\_label)$
$inner\_labels'(label\_object) = inner\_labels(target\_label)$
$inner\_labels'(certificate) = inner\_labels(instigator)$
$role'(\!|\{certificate, label\_object\}|\!) = \{inner\_role(instigator)\}$
$g'(\!|\{parent\}|\!) = g(\!|\{parent\}|\!) \cup \{label\_object\}$
$g'(\!|\{label\_object\}|\!) = \{certificate\}$
$g'(\!|\{certificate\}|\!) = \{target\_label\}$
$data'(\!|\{instigator\}|\!) = data(\!|\{instigator\}|\!) \cup \{USER\_REQUEST\_DOWNGRADE\_OK\}$
$data'(\!|\{certificate\}|\!) = data(\!|\{target\}|\!)$
$data'(\!|\{label\_object\}|\!) = \emptyset$
$inner\_role'(\!|\{certificate, label\_object\}|\!) = \{inner\_role(instigator)\}$

$BAD\_USER\_REQUEST\_DOWNGRADE \triangleq \Phi STATE_{[USER\_REQUEST\_DOWNGRADE\_NOT\_OK/RESULT]}$

$USER\_REQUEST\_DOWNGRADE \triangleq BAD\_USER\_REQUEST\_DOWNGRADE \oplus GOOD\_USER\_REQUEST\_DOWNGRADE$

user_request_downgrade keeps USER_REQUEST_DOWNGRADE

basic_types :Module	labels :Module	basic_state :Module

The Initiate Downgrade Command

The actual commitment of a downgrade requires the creation of a new alpha object. Thus this transtion must once again be modelled as a two part command involving the garbage collector.

ΦINIT_USER_DOWNGRADE_ALPHA

$\Delta STATE$
$certificates : \mathbb{P}\, E$
$target\_label : E$
$label : SECURITY\_LABELS$

$g' = g$
$outer\_labels' = outer\_labels$
$entitytype' = entitytype$
$role' = role$
$\{instigator, GC\} \lhd data' = \{instigator, GC\} \lhd data$
$inner\_labels' = inner\_labels \oplus \{GC \mapsto label\}$
$inner\_role' = inner\_role \oplus \{GC \mapsto role(instigator)\}$

INIT_USER_DOWNGRADE_ALPHA_REQUEST
ΦINIT_USER_DOWNGRADE_ALPHA

$entitytype(instigator) = User$
$entitytype(target\_label) = Label\_Object$
$\forall\, e : certificates \bullet (entitytype(e) = Certificate \land$
$g(\{e\}) = \{target\_label\} \land$
$data(\{e\}) = data(certificates))$
$inner\_labels(certificates) = \{label\}$

GOOD_INIT_USER_DOWNGRADE_ALPHA
INIT_USER_DOWNGRADE_ALPHA_REQUEST
$INIT\_USER\_DOWNGRADE\_ALPHA\_OK : DATUM$

$certificates \subseteq g(\{instigator\})$
$\forall\, e : certificates \cup \{target\_label, instigator\} \bullet$
$(outer\_labels(e)).conflict \subseteq role(certificates)$
$data'(\{instigator\}) = data(\{instigator\}) \cup \{INIT\_USER\_DOWNGRADE\_ALPHA\_OK\}$
$data(certificates) \subseteq data'(\{GC\})$

$BAD\_INIT\_USER\_DOWNGRADE\_ALPHA \triangleq \Phi STATE_{[INIT\_USER\_DOWNGRADE\_ALPHA\_NOT\_OK/RESULT]}$

$INIT\_USER\_DOWNGRADE\_ALPHA \triangleq BAD\_INIT\_USER\_DOWNGRADE\_ALPHA \oplus GOOD\_INIT\_USER\_DOWNGRADE\_ALPHA$

init_user_downgrade_alpha keeps INIT_USER_DOWNGRADE_ALPHA

basic_types :Module	labels :Module	basic_sta' :Module

The GC Execution of the Downgrade Command

This command creates the new alpha object and initialises it with the data from the certificates. It links this into the existing label object and adjusts its inner labels.

$\Phi USER\_DOWNGRADE\_ALPHA$

$\Delta STATE$
$target\_label : E$
$new\_target : E$

$\{target\_label, new\_target\} \lhd g' = \{target\_label, new\_target\} \lhd g$
$\{new\_target\} \lhd outer\_labels' = \{new\_target\} \lhd outer\_labels$
$entitytype' = entitytype \oplus \{ new\_target \mapsto Alpha \}$
$role' = role \oplus \{new\_target \mapsto inner\_role(instigator)\}$
$\{instigator, new\_target\} \lhd data' = \{instigator, new\_target\} \lhd data$
$\{target\_label, new\_target\} \lhd inner\_labels' = \{target\_label, new\_target\} \lhd inner\_labels$
$inner\_role' = inner\_role \oplus \{new\_target \mapsto inner\_role(instigator)\}$

$USER\_DOWNGRADE\_ALPHA\_REQUEST$
$\Phi USER\_DOWNGRADE\_ALPHA$

$dom(g \rhd \{new\_target\}) = \{GC\}$
$instigator = GC$
$entitytype(target\_label) = Label\_Object$

$GOOD\_USER\_DOWNGRADE\_ALPHA$
$USER\_DOWNGRADE\_ALPHA\_REQUEST$
$USER\_DOWNGRADE\_ALPHA\_OK : DATUM$

$g'(\!|\{target\_label\}|\!) = \{new\_target\}$
$g'(\!|\{new\_target\}|\!) = \emptyset$
$outer\_labels'(new\_target) = inner\_labels(instigator)$
$data'(\!|\{instigator\}|\!) \subseteq data(\!|\{instigator\}|\!) \cup \{USER\_DOWNGRADE\_ALPHA\_OK\}$
$data'(\!|\{new\_target\}|\!) \subseteq data(\!|\{instigator\}|\!)$
$inner\_labels'(target\_label) = inner\_labels(instigator)$
$inner\_labels'(new\_target) = inner\_labels(instigator)$

$BAD\_USER\_DOWNGRADE\_ALPHA \triangleq \Phi STATE_{[USER\_DOWNGRADE\_ALPHA\_NOT\_OK/RESULT]}$

$USER\_DOWNGRADE\_ALPHA \triangleq BAD\_USER\_DOWNGRADE\_ALPHA$
$\oplus GOOD\_USER\_DOWNGRADE\_ALPHA$

user_downgrade_alpha keeps USER_DOWNGRADE_ALPHA

REPORT DOCUMENTATION PAGE

DRIC Reference Number (if known) .....

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the field concerned must be marked to indicate the classification eg (R), (C) or (S).

Originators Rererence/Report No.	Month	Year
REPORT 91022	APRIL	1991

Originators Name and Location
RSRE, St Andrews Road
Malvern, Worcs WR14 3PS

Monitoring Agency Name and Location

Title
AN EXAMPLE MACHINE USED FOR DEVELOPING A PROOF STRATEGY FOR SECURE SYSTEMS

Report Security Classification	Title Classification (U, R, C or S)
UNCLASSIFIED	U

Foreign Language Title (in the case of translations)

Conference Details

Agency Reference	Contract Number and Period
	SLS 42c/720
Project Number	Other References

Authors	Pagination and Ref
TERRY, P F	56

Abstract

This report describes a machine which is an abstraction of the archetypal Command, Control, Communications and Information (C^3I) system which system developers meet in procurement requests, operational requirements, invitations to tender, etc, from government and military agencies.

The purpose of this report is to set the scope of complexity of structure, functionality and policy which we believe the SMITE approach to secure systems development can encompass. It thus provides background and motivation for future research and encourages those involved in secure systems procurement to investigate further the SMITE approach.

The Abstract Machine is first described in English with pictures and subsequently in the Z specification language.

Abstract Classification (U,R,C or S)
U

Descriptors

Distribution Statement (Enter any limitations on the distribution of the document)

UNLIMITED